병렬가상기계(PVM)프로그람 작성지도서

국가과학원 콤퓨터과학연구소

주체 98(2009)년 4 월

차　　　례

1. 프로그람작성방법

병렬가상기계체계를 리용하는 병렬응용프로그람을 개발하자면 적어도 다중처리기에서와 분산기억형다중처리기에서의 병렬프로그람작성방법을 알아야 한다.

기본적인 수법들은 알고리듬개발이나 프로그람의 론리적인 개념에서 직렬프로그람작성방법과 류사하다.

그러나

1. 과제관리측면에서 특히 동적인 프로쎄스생성, 이름달기, 주소화방법
2. 실제적인 계산단계전에 진행하는 초기화단계
3. 립도의 선택
4. 이종성

측면에서 특징적인 차이점들이 존재한다.

여기서는 병렬가상기계에서의 병렬프로그람작성방법에 대하여 서술하고 기능과 성능에 영향을 줄수 있는 인자들에 대하여 설명한다.

1.1 일반적인 병렬프로그람작성모형

병렬가상기계체계를 리용한 병렬계산을 위한 계산모형은 계산과제들의 선정 및 분할방식에 기초하여 3가지 방식으로 진행한다.

① 가장 공통적이고도 중요한 병렬가상기계체계의 계산모형은 “무리계산” 모형이다.

이 모형에서는 전형적으로 매 과제들이 서로 밀접히 결합되여있다.

매 과제들은 같은 코드를 실행하고 서로 다른 자료부분을 처리하며 보통 중간결과들을 주기적으로 교환한다.

이 모형은 두가지 부류로 나눌수 있다.

- 주-종속모형

이 모형에서 주프로그람이라고 부르는 조종프로그람은 프로쎄스의 실행, 초기화, 결과의 집합 등의 기능을 수행한다.

종속프로그람은 실제적인 계산부분을 처리한다. 그것들은 주프로그람으로부터 작업부하를 배정받거나(정적으로 혹은 동적으로)자기에게 분배된 부분을 처리한다.

- 마디전용모형

여기서는 한개의 프로그람이 여러개의 실체로 실행된다. 한개의 프로그람은(이것은 대체로 수동적으로 초기화된다.) 계산부분과 함께 비계산응답기능도 가지고있다.

② 병렬가상기계에서 제공하는 두번째 모형은 "나무형"계산모형이다.

이 알고리듬에서 프로쎄스들은(보통 계산과정에 동적으로 진행된다.) 나무와 류사한 방식으로 실행되므로 나무와 같은 어미-자식관계를 가진다(별형구조와 류사한 혼합형계산모형과는 반대이다.).

이 모형은 잘 리용되지는 않지만 전체 작업부하가 사전에 잘 알려지지 않은 경우, 실례로 가지한정알고리듬, 재귀적인 "점유 및 포획"알고리듬과 같은 응용들에서 대단히 유익하다.

③ 세번째모형은 "혼합형"모형이라고 부른다.

이것은 "나무"모형과 "무리계산"모형의 조합으로 볼수 있다.

이 모형은 임의의 시동구조를 가지고있다. 즉 응용프로그람실행의 임의의 시점에서 프로쎄스관련구조는 임의로 될수 있으며 그라프는 변경된다.

우리는 이 세가지 정식화들이 비록 통신위상에 따라 변할수는 있다 해도 프로쎄스관계의 기초로 된다는것을 강조한다.

그럼에도 불구하고 이 세가지 전략들에서 임의의 프로쎄스들은 다른 프로쎄스들과 통신 및 동기화를 진행할수 있다.

또한 모형의 선택은 응용프로그람에 의존하며 병렬화해야 할 때 프로그람의 구조에 가장 알맞는것을 선택해야 한다.

1.1.1 무리형 계산모형

무리계산모형은 일반적으로 세단계로 구성된다. 첫단계는 프로쎄스그룹의 초기화단계이다. 마디전용계산인 경우 이 단계에서는 작업부하분배와 마찬가지로 그룹정보와 파라메터들을 준다.

두번째 단계는 계산단계이다. 세번째 단계는 결과의 모으기와 출구단계이다. 이 단계에서 프로쎄스그룹은 해체되며 완료된다. 잘 알려진 만델브로트(Mandelbrote)모임계산을 실례로 주-종속모형을 아래에서 설명한다.

이 문제는 병렬화가 대단히 힘든 분야의 문제이다. 계산 그 자체는 함수값이 지정된 값에 도달하였거나 발산하기 시작할 때까지 복소수평면에 있는 점들에 재귀함수를 적용한다. 이 조건에 따라 평면에 있는 매 점이 그라프적으로 표시된다.

함수결과는 점(다른 점들과는 독립이다.)의 초기값에 의존하므로 문제를 완전히 독립적인 부분들로 가를수 있으며 매개 부분에 알고리듬을 적용하고 계산결과들을 단순한 합성기법을 리용하여 묶을수 있다.

그러나 이 수법은 동적인 부하균등을 허용하기때문에 처리기들은 균등한 작업부하를 가질수 있다. 응용프로그람의 주-종속구조를 그림 1에 보여준다.

만델브로트계산알고리듬

```
{초기 배치}
for i := 0 to NumWorkers - 1
pvm_spawn(<worker name>) {작업기 i를 시동}
pvm_send(<worker tid>, 999) {작업기 i에 과제를 보낸다.}
endfor
{수신-송신}
while (WorkToDo)
pvm_recv(888) {결과접수}
pvm_send(<available worker tid>, 999)
{다음 과제를 가능한 처리기에 보낸다.
 결과 현시}
endwhile
{결과 모으기}
for i := 0 to NumWorkers - 1
pvm_recv(888) {결과접수}
```

```
pvm_kill(<worker tid i>) {작업기 i를 완료}
display result
endfor


{처리기들에서의 계산}

while (true)
pvm_recv(999) {과제접수}
result := MandelbrotCalculations(task) {결과계산}
pvm_send(< tid>, 888) {결과를 주처리기에게 보낸다.}
endwhile
```
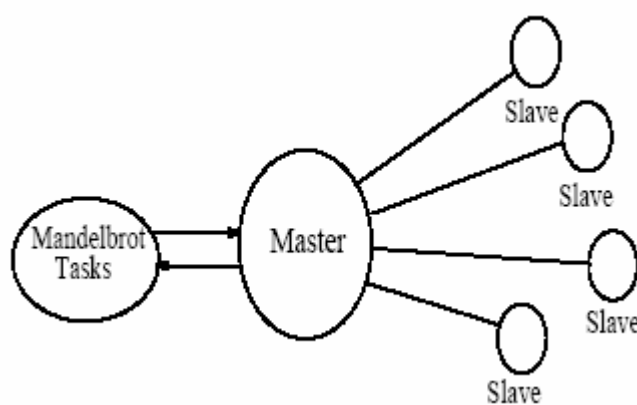
그림 1 주/종속모형

우에서 서술된 주-종속 실례는 종속과제들사이의 통신은 고려하지 않았다.

대다수의 무리계산모형들에서는 계산과제들사이에 통신을 요구한다.

우리는 캐논의 알고리듬을 리용한 행렬곱하기실례를 가지고 그러한 응용들의 구조를 설명한다(류사한 알고리듬에 대해서는 다음 장에서 구체적으로 서술한다.).

그림 2의 행렬곱하기실례에서는 행렬의 부분블로크들을 국부적으로 곱하고 행렬 B의 부분블로크들을 렬방향으로 밀고 그 다음에 행렬 A의 부분블로크들을 행방향으로 방송한다.

```
{행렬곱하기 알고리듬}
{처리기 0은 다른 프로세스들을 시동한다.}
if (<my processor number> = 0) then
   for i := 1 to MeshDimension*MeshDimension
      pvm_spawn(<component name>, . .)
   endfor
endif
forall processors Pij, 0 <= i, j < MeshDimension
  for k := 0 to MeshDimension-1
     if myrow = (mycolumn+k) mod MeshDimension
      {Send A to all Pxy, x = myrow, y <> mycolumn}
      pvm_mcast((Pxy, x = myrow, y <> mycolumn), 999)
   else
      pvm_recv(999) {Receive A}
   endif
    {Multiply. Running totals maintained in C.}
   Multiply(A, B, C)
    {Roll.}
    {Send B to Pxy, x = myrow-1, y = mycolumn}
  pvm_send((Pxy, x = myrow-1, y = mycolumn), 888)
  pvm_recv(888) {Receive B}
 endfor
endfor
```

First 4 Steps of Pipe-Multiply-Roll on a 3x3 Mesh-Connected Machine

Step 1: First "pipe"

Step 2: Multiply temp matrix and matrix B

Step 3: First "roll"

Step 4: Second "pipe"
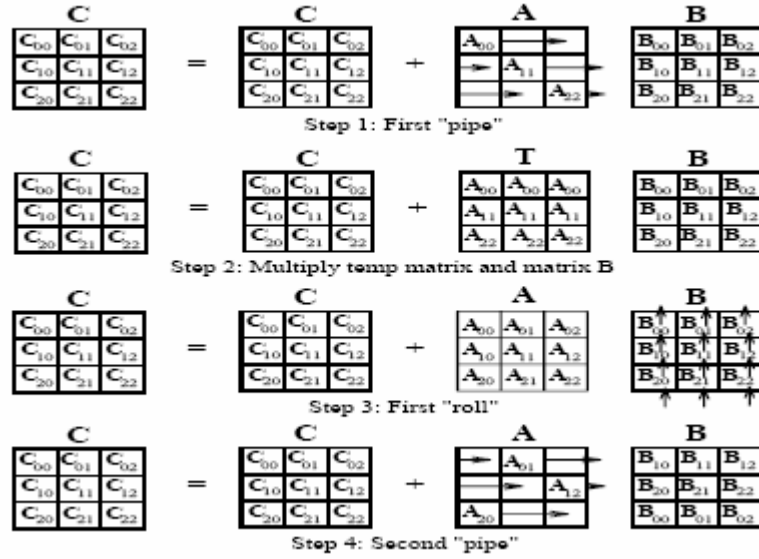
그림 2 일반적인 무리계산

1.1.2 나무형 계산모형

앞에서 서술한바와 같이 나무형계산들은 전형적으로 나무와 류사한 프로쎄스구조를 가지고있으며 또한 많은 실체들에서 통신류형들을 확정한다.

이 모형을 설명하기 위하여 우리는 다음과 같은 병렬분류알고리듬을 고찰한다. 분류해야 할 목록은 한 프로쎄스가 가지고있다. 이 프로쎄스는 두번째 프로쎄스를 생성하며 목록의 절반을 그 프로쎄스에 보낸다.

이 시점에서 매개 프로쎄스는 각각 한개의 자식프로쎄스를 생성하고 이미 절반으로 나누어진 목록의 절반을 그것들에 보낸다. 이 과정은 적절한 깊이의 나무가 구성될 때까지 계속된다.

매개 프로쎄스는 목록에서 자기에게 해당한 부분을 독립적으로 분류한다. 매개 마디에서 중간병합과정들이 이루어지고 다음에 나무의 변들을 따라 우로 올라가면서 병합과정이 이루어진다. 이 알고리듬은 나무계산에 대하여 구체적으로 설명한다. 작업부하는 실행과정에 알려진다. 프로쎄스를 묘사하는 과정을 그림 3에 보여주었다.

알고리듬은 다음과 같다.

```
{ 방송형 나무류형에 기초한 목록의 생성 및 나누기 }
for i := 1 to N,   such that 2^N = NumProcs
    forall processors P such that P < 2^i
   pvm_spawn(...) {process id P XOR 2^i}
   if P < 2^(i-1) then
      midpt: = PartitionList(list);
       {목록 [0..midpt]을  P XOR 2^i에 보낸다.}
      pvm_send((P XOR 2^i), 999)
      list := list[midpt+1..MAXSIZE]
  else
      pvm_recv(999) {목록접수}
  endif
 endfor
endfor

{ 나머지 목록의 분류 }
Quicksort(list[midpt+1..MAXSIZE])
{ 분류된 부분목록들을 모으기 및 병합한다. }
  for i := N downto 1,  such that 2^N = NumProcs
      forall processors P such that P < 2^i
        if P > 2^(i-1) then
          pvm_send((P XOR 2^i),  888)
          {Send list to P XOR 2^i}
        else
           pvm_recv(888) {receive temp list}
           merge templist into list
        endif
      endfor
endfor
```
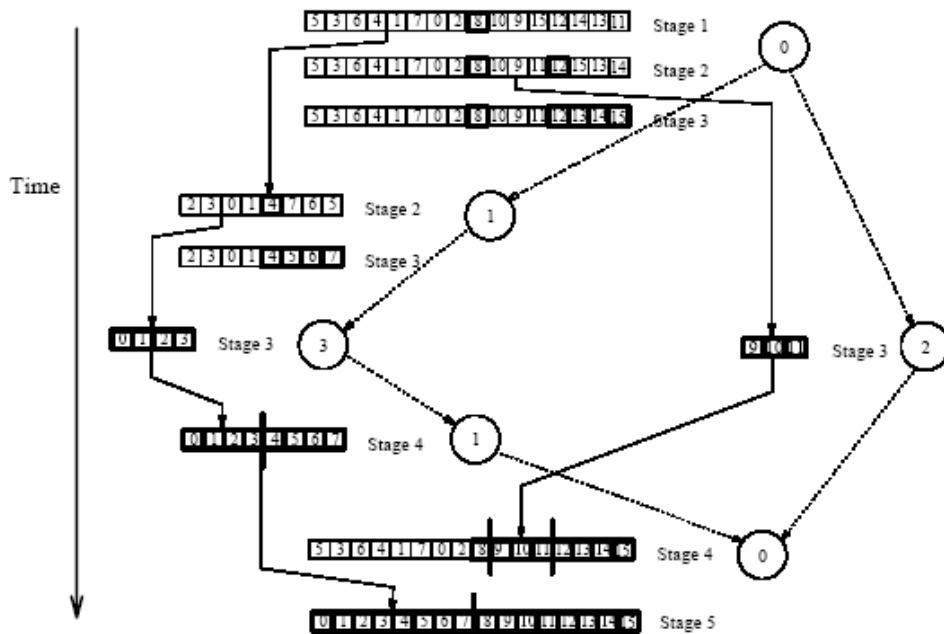
그림 3 나무형 계산실례

1.2 작업부하의 분배

앞절에서는 프로쎄스구조에 따르는 공통적인 병렬프로그람작성모형에 대하여 서술하고 병렬가상기계체계의 측면에서 직관적인 실례를 가지고 설명하였다.

이 절에서는 작업부하분배문제를 취급하고 프로쎄스구조를 확정하며 분산기억형콤퓨터체계에서의 병렬계산에서 리용되는 몇가지 공통적인 모형들에 대하여 설명한다.

첫째: 자료분배와 나누기라는것은 모든 문제가 한개이상의 자료구조우에서 계산과 통신을 가지고있으며 또 이 자료구조들은 그 우에서 나누어지고 계산된다는것을 의미한다.

둘째: 기능분할이라는것은 계산과제를 서로 다른 기능이나 함수들로 나눈다는것을 의미한다.

기능적으로 병렬가상기계계산모형은 기능분할(기능적으로 서로 다른 과제들은 서로 다른 연산들을 수행한다.)과 자료분할(동등한 과제들이 자료의 서로 다른 부분을 처리한다.)을 둘다 지원하고있다.

1.2.1 자료분할

자료분할의 단순한 실례로서 두개의 벡토르 A[1..N]와 B[1..N]의 더하기를 고찰하자. 결과는 C[1..N]에 보관한다.

우리는 이 문제를 푸는데 p개의 프로쎄스들이 있으며 매 프로쎄스들에 N/P개의 자료요소들이 분배되며 결과벡토르들도 N/P개의 요소들로 계산된다고 가정한다.

이 자료분할은 정적으로 할수도 있고(여기서 매개 프로쎄스는 N 과 P의 값을 가지고 사전에 자기가 처리해야 할 작업량을 알고있다.) 혹은 동적으로 할수도 있다.

동적자료분할의 경우 조종프로쎄스(실례로 주프로쎄스)는 프로쎄스들이 연산을 끝날 때마다 작업부하들을 분배한다.

이 두가지 방안들의 원리적인 차이는 계획화에 있다. 정적인 계획화에서는 매개 프로쎄스들의 작업부하가 고정되여있고 동적인 계획화에서는 프로쎄스들의 작업부하가 시간에 따라 변한다.

대다수의 다중처리기환경에서는 벡토르더하기실례에서와 같은 문제들에 대해서 정적인 계획화가 더 효과적이다. 그러나 일반적으로 병렬가상기계환경에서는 정적계획화가 항상 효과적인것은 아니다. 그 리유는 콤퓨터들을 망으로 련결한 국부망에서의 병렬가상기계환경이 외부환경의 영향을 많이 받기때문이다.

그러므로 정적으로 계획화되고 자료가 분할된 문제는 한개 이상의 프로쎄스들로 실행할수 있는데 다른 계획화방법들보다 더 빨리 혹은 더 천천히 실행될수 있다.

이런 경우는 병렬가상기계체계에 있는 처리기들이 이종성을 가질 때, 즉 CPU속도가 다르고 기억기용량이 차이나며 체계의 속성이 다른 경우에도 일어날수 있다. 지어 작은 벡토르더하기문제를 실행할 때에조차 무시할수 없는 문제는 입구와 출구이다.

다시말하여 우에서 서술한 프로쎄스들이 자기의 작업부하를 어떻게 접수하며 그것들이 결과벡토르를 가지고 무엇을 하는가 하는 질문에 대한 대답은 응용프로그람과 구체적인 실행환경에 따라 달라진다.

그러나 일반적으로

1) 개별적인 프로쎄스들은 자기 자체의 자료를 란수를 리용하든가 정적으로 알려진 값들을 리용하여 내부적으로 생성한다. 이것은 다만 특별한 경우에 혹은 프로그람검사목적을 위해서만 가능하다.

2) 개별적인 프로쎄스들은 외부장치로부터 자기의 자료들을 독립적으로 입력한다. 이 방법은 대다수의 경우에 의미가 있지만 이것은 병렬 입출구가 제공될 때만 가능하다.

3) 조종프로쎄스는 개별적인 자료부분모임들을 매 프로쎄스에 보낸다. 이것은 특히 병렬입출구기능을 지원하지 않는 경우에 특별히 효과적이다. 이 방법은 또한 입구자료부분모임들이 같은 응용프로그람내에서 이전의 계산결과로부터 나오는 경우에 편리하다.

개별적으로 작업부하를 분배하는 세번째 방법은 계산과정에 호상작용이 거의 없거나 존재하지 않는 응용들을 동적으로 계획화하는 경우에도 부합된다.

그러나 일반적으로 알고리듬들에서는 자료의 중간결과들을 교환하므로 이 수법들에 의해서는 다만 자료의 초기분배만이 실현된다. 실례로 그림 2에서 서술된 자료분할방법을 고찰하자.

두 행렬 A 와 B를 곱하기 위하여 주-종속 혹은 마디전용모형을 리용하여 프로쎄스들의 그룹을 먼저 생성한다.

이 프로쎄스들의 모임은 그물을 형성하며 곱해야 할 행렬들도 그물을 형성하면서 부분블로크들로 나누어진다.

행렬 A 와 B의 부분블로크들은 우에서 서술한 자료분배와 작업부하분배전략들중 하나를 리용하여 대응하는 프로쎄스에 배치된다.

계산도중에 부분블로크들은 프로쎄스들사이에서 교환되여야 하므로 그림에서 보여준바와 같이 원래의 분배표를 보낸다.

그러나 계산의 마감에 행렬부분블로크들은 개별적인 프로쎄스들에 위치하고있으며 프로쎄스그물과 결과행렬 C의 대응하는 위치에 있다.

1.2.2 기능분할

병렬가상기계의 병렬실행환경에서는 전체 작업부하를 연산단위로 나누어 실현할수도 있다. 이러한 형태의 분할에 대한 가장 명백한 실례는 전형적인 프로그람실행의 세가지 단계 즉 입구, 처리, 결과출구와 관련된다.

기능분할에서 그러한 응용은 세개의 개별적이며 명백한 프로그람들로 구성되며 매개는 세 단계들중의 하나로 된다. 병렬성은 세개의 프로그람들을 동시에 실행시키고 그것들사이에 관흐름을 실현하므로써 얻어진다.

그러나 그러한 모형에서도 매 단계에 자료병렬성이 존재한다. 기능분할의 개념은 우에서 서술한 실례로 례증할수 있다.

그러나 일반적으로 이 의미는 기능별로 작업부하를 분배하거나 분할한다는것을 표시하는데 리용한다.

일반적으로 응용프로그람들은 여러가지 서로 다른 알고리듬들을 포함하며 때로는 같은 자료(MPSD-Multiple Program Single Data)에서 때로는 련속자료전송형태로, 때로는 비구조적인 통신형태로 존재할수 있다.

우리는 다중으로 호상련결되고 호상작용하는, 기능적으로는 분할된 비행기의 가상모의를 통하여 일반적인 기능분할모임을 설명한다. 이 실례에 대한 도식을 그림 4에 보여준다.

그림에서 그라프에 있는 매개 마디 혹은 원은 기능적으로 분할된 응용프로그람의 토막을 보여준다. 입구함수들은 다른 함수들인 2~6에 개별적인 문제파라메터들을 분배하며 매 부분알고리듬들을 수행하는 프로그람들에 대응하는 프로쎄스들을 생성한다.

여러 함수들에 같은 자료가 전송될수 있으며(실례로 두개의 날개함수인 경우에)주어진 함수자체에 필요한 자료가 전송될수도 있다. 계산을 진행한 후에 이 함수들은 계산의 시작시점에서 혹은 결과가 필요하여 생성된 함수 7, 8, 9에 중간결과 혹은 최종결과를 보낸다. 이 모형은 자료 및 조종의존성과 마찬가지로 기능에 따라 응용프로그람을 분할하는 개념을 서술하고있다.
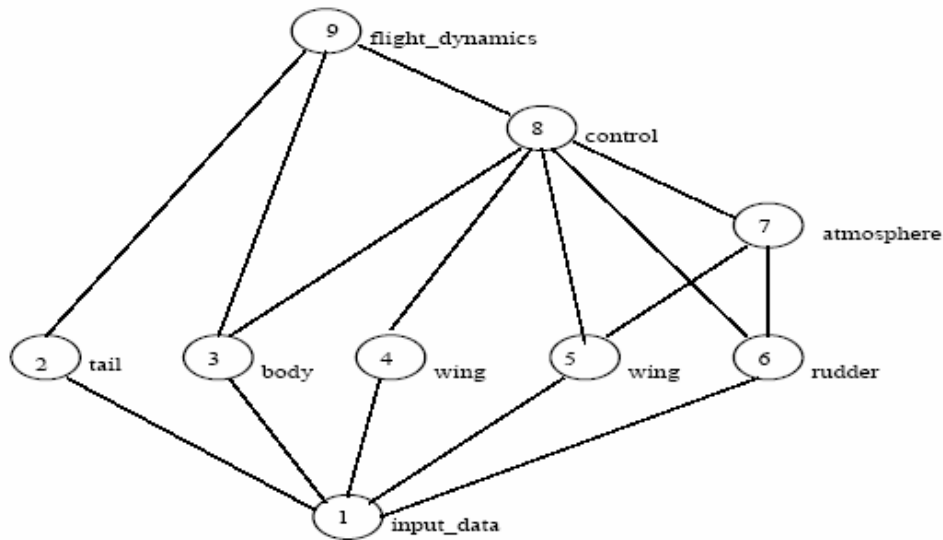
그림 4 기능분할실례

병렬성은 함수 2~6에서와 같이 모듈들을 병렬로, 독립적으로 실행시키거나 의존성서술에 따라 즉 함수 1, 6, 8, 9순서로 모듈들을 실행시키는 방법으로 두가지 측면에서 실현될수 있다.

1.3 현존 응용프로그람들을 병렬가상기계에 이식

병렬가상기계체계를 리용하자면 응용프로그람들을 두 단계로 개발하여야 한다.

먼저 응용프로그람의 알고리듬을 분산기억형병렬프로그람형식으로 개발하여야 한다. 이 단계는 다른 분산기억다중처리기에서와 마찬가지로 병렬가상기계체계에서도 필요하다. 실제적인 병렬화결심들은 다음의 두 단계로 귀착된다. 즉 구조와 관련된것과 효과성과 관련된것이다. 응용프로그람병렬화에서 구조적인 결심을 하는데서 중요한것은 리용하여야 할 모형을 선택하는것이다(즉 무리계산 대 나무형계산과 자료분할 대 기능분할). 분산기억환경에서 병렬화를 진행할 때 효과성과 관련되는것은 일반적으로 통신의 량과 빈도수를 최소로 하는것이다. 병렬가상기계환경은 망에 기초하고있으므로 일반적으로 립도를 크게 하는것이 효과적이다. 이런 측면에서 프로그람의 병렬화는

병렬가상기계에서나 하드웨어다중처리기들을 포함한 다른 분산기억환경에서 아주 류사하다.

다음으로 현재의 직렬프로그람이나 병렬프로그람을 처음부터 병렬화할수 있다.

우의 두 경우에는 먼저 이미 서술된것으로부터 혹은 적절한 병렬알고리듬을 개발하고 그 다음 적당한 프로그람작성언어로 이 알고리듬을 코드화한다.

현존 병렬프로그람들을 병렬화하는것은 일반적인 원칙에 따라 진행하는데 먼저 순환들로 분해하고 제일 바깥순환부터 시작하여 실행하는것이다. 이 처리에서 중요한것은 의존성들을 검출하고 의존성이 보존되도록 하면서 순환들을 분해하는것이다. 이 병렬화원리는 많은 책들과 론문들에서 서술되였다.

그러나 순차프로그람을 병렬프로그람으로 넘기는 실천적이고도 독특한 방법들을 서술한 책은 거의 없다.

현존 병렬프로그람들은 공유기억 혹은 분산기억모형에 기초하고있다.

현존 공유기억형 프로그람들을 병렬가상기계에 넘기는것은 직렬코드를 변환하는것과 아주 류사하다.

그것은 공유기억형프로그람들이 벡토르 혹은 고리준위병렬화에 기초하고있기때문이다. 공유기억과제인 경우에 처음으로 해야 할 처리는 동기화시점들을 찾고 이것들을 통보전달기능으로 바꾸는것이다.

현존 분산기억형프로그람을 병렬가상기계로 넘기기 위해서는 먼저 한가지 병행성구조모임을 다른 모임으로 변환한다.

일반적으로 현존 분산기억병렬프로그람들은 p4나 Express와 같은 하드웨어 다중처리기나 다른 망환경을 위하여 작성되여있다. 두 경우에 프로쎄스관리와 관련하여 수정하여야 할 중요한 문제가 있다.

실례로 인텔계렬처리기들에서는 호상작용하는 쉘지령들을 리용하여 프로쎄스들을 시동시키는것이 보통이다.

병렬가상기계에서는 그러한 모형을 프로쎄스를 생성하는 능력을 가진 주프로그람 혹은 마디프로그람으로 바꿀수 있다.

호상작용측면에서 고려하여야 할 공통적인 문제는 여러가지 프로그람작성환경에서의 통보전달호출문제이다.

이러한 문맥에서 병렬가상기계와 다른 체계들과의 기본 차이점은

1) 프로쎄스관리와 프로쎄스취급방법
2) 가상기계의 구성과 재구성, 병렬응용프로그람실행에서 그의 영향
3) 서로 다른 구성과 자료표현수법을 취급하는데로부터 생기는 통보의 이종성
4) 신호기수법과 과제계획화방법과 같은 고유한 특징

2. 병렬가상기계의 리용자대면부

이 장에서는 병렬가상기계의 함수들에 대하여 간단히 설명한다. 실례로 통보전달에 대한 절에서는 한 병렬가상기계과제로부터 다른 과제에로 자료들을 주고받는 모든 함수들을 설명하며 또한 병렬가상기계의 통보전달추가선택들에 대하여 설명한다.

매개 함수에서는 또한 C와 포트란 사용실례들을 주었다.

병렬가상기계에서 모든 병렬가상기계과제들은 국부 pvmd가 제공하는 옹근수식별자로써 식별된다.

다음에서 이러한 과제식별자를 TID라고 부른다. 이것은 Unix체계에서 리용되는 프로쎄스번호와 같으며 TID의 값이 리용자에게 특별한 의미를 가지지 않으므로 리용자에게는 불투명하다.

사실상 병렬가상기계는 그 자체의 내부사용을 의해 정보를 TID로 해신한다. 모든 병렬가상기계함수들은 C언어로 작성하였다. C++언어로 작성된 프로그람들도 병렬가상기계서고에 련결할수 있다.

포트란응용프로그람들은 병렬가상기계체계가 지원하고있는 포트란77 대면부를 통하여 이 함수들을 호출할수 있다. 이 대면부는 변수들을 번역하며 이 변수들은 포트란서술로 넘긴다.

이 대면부는 또한 포트란문자렬표기와 여러가지 이름달기약속들을 고려하여 여러가지 포트란콤파일러들이 C함수들을 호출할수 있게 한다.

병렬가상기계통신모형에서는 임의의 과제가 다른 병렬가상기계과제에 통보를 보낼수 있으며 그러한 통보들의 크기나 개수에는 제한이 없다고 가정한다.

모든 처리기들은 완충기억을 제한하는 물리적인 기억한계를 가지고 있기때문에 통신모델은 그 자체가 개별적인 처리기들의 제한성을 고려하지 않으며 충분한 기억기가 있다고 가정한다.

병렬가상기계통신모델은 비동기적인 봉쇄전송, 동기적인 봉쇄수신, 비봉쇄수신기능들을 제공하고있다. 여기서 봉쇄송신이라는것은 송신완충기가 비여 다시 사용하게 될 때마다 되돌린다는것이다. 비봉쇄전송은 송신이 귀환하기전에 수신자가 자기에게 오는 정보를 받을것을 요구하지 않는다.

병렬가상기계에는 과제로부터 과제에로 자료를 직접 전송할것을 요구하는 추가선택들이 있다. 이 경우에 만일 통보가 크다면 송신자는 수신자가 일치한 수신을 받을 때까지 봉쇄될수도 있다.

비봉쇄수신은 자료가 도착하지 않아도 즉시 귀환되지만 봉쇄수신은 자료가 수신완충기에 들어왔을 때만 귀환된다.

이러한 점대점통신에 보충적으로 이 통신모델에서는 어떤 과제모임에로의 다중통신과 리용자가 정의한 과제그룹에로의 방송통신을 지원하고있다.

또한 리용자가 정의한 과제들사이에서 대역적인 최대값구하기, 대역적인 더하기 등을 수행하는 여러가지 기능들을 가지고있다. 병렬가상기계모델은 통보의 순서가 보존된다는것을 담보한다.

만일 과제 1이 과제 2에게 통보 A를 보내고 다음에 과제 1이 과제 2에 통보 B를 보낸다면 통보 A는 통보 B전에 도착한다. 또 과제 2가 접수하기전에 두 통보들이 도착한다면 항상 통보 A가 먼저 처리된다.

통보완충기들은 동적으로 할당한다. 그러므로 송신하거나 수신할수 있는 최대통보크기는 주어진 처리기에서 리용할수 있는 기억기의 크기에 의존한다.

병렬가상기계체계에는 유일하게 구축된 한정된 흐름조종이 있다. 병렬가상기계는 오는 통보들의 전체 크기가 리용가능한 기억기량을 초과하는 경우 필요한 기억기를 얻을수 없다는 오유를 내보내지만 병렬가상기계는 다른 과제들이 이 처리기에 전송을 중지하라고는 통보하지 않는다.

2.1 프로쎄스조종

```
int tid = pvm_mytid( void )
call pvmfmytid( tid )
```

pvm_mytid()함수는 이 프로쎄스의 TID를 돌려주며 여러번 호출할수 있다. 만일 이것이 첫 호출이라면 그것을 병렬가상기계에 기록한다. 임의의 병렬가상기계체계호출은 만일 그 과제가 호출전에 체계에 기록되지 않았다면 과제를 체계에 기록한다.

그러나 체계에 과제를 기록하기 위하여 pvm_mytid()함수를 먼저 호출하는것이 일반적이다.

```
int info = pvm_exit( void )
call pvmfexit( info )
```

pvm_exit()함수는 국부 pvmd에게 이 프로쎄스가 체계를 떠난다는것을 알려준다. 이 함수는 프로쎄스를 죽이지 않으며 다른 Unix프로쎄스들과 같이 과제들을 계속 실행한다.

일반적으로 리용자들은 자기의 C프로그람을 끝내기전에 혹은 포트란 프로그람에서 STOP지령전에 pvm_exit()함수를 리용한다.

```
int numt = pvm_spawn( char *task, char **argv, int
            flag, char *where, int ntask, int *tids )
call pvmfspawn( task, flag, where, ntask, tids, numt )
```

pvm_spawn()함수는 가상기계에서 병렬가상기계응용프로그람들을 생성하는데 리용된다. argv는 NULL로 끝나는 과제변수들의 배렬에 대한 지적자이다. 과제가 변수들을 가지지 않는다면 argv는 NULL로 된다. flag 변수는 추가선택들을 지적하는데 리용되는데 그의 의미는 다음과 같다.

값	추가선택	의미
0	PvmTaskDefault	프로쎄스들을 어디서 실행시키겠는가를 결정한다.
1	PvmTaskHost	where변수는 실행할 처리기에 고유한 값이다.
2	PvmTaskArch	where변수는 실행할 처리기의 PVM_ARCH이다.
4	PvmTaskDebug	과제들을 오유제거기우에서 실행시킨다.

8　PvmTaskTrace　　추적자료를 생성한다.
16　PvmMppFront　　과제들을 MPP의 앞단마디에서 실행시킨다.
32　PvmHostCompl　　where에 있는 처리기모임을 보충한다.

이 이름들은 pvm/include/pvm.h에 미리 정해져있다. 포트란에서는 모든 이름들이 파라메터지령들로 미리 정의되여있으며 이것은 pvm/include/fpvm.h 파일에서 찾을수 있다.

PvmTaskTrace는 PVM3의 새로운 특징이다. 이 추가선택을 설정하여 생성된 과제들에 대한 추적사건들을 생성한다. 다른 경우 pvm_setopt()함수로 추적사건들을 어디로 보내야 하는가를 지적해야 한다. 함수가 귀환되였을 때 numt에는 실제로 생성된 과제들의 개수 혹은 오유가 발생한 경우 오유코드가 들어있다.

만일 과제들이 생성되였다면 pvm_spawn()함수는 생성된 과제들의 배렬 tids를 돌리며 일부 과제들이 시동되지 않았다면 과제벡토르의 해당 위치에 오유코드가 배치된다.

pvm_spawn()호출은 또한 다중처리기에서 과제들을 실행할수 있다.

InteliPSC/860인 경우에는 다음의 제한이 있다. 매개의 생성호출은 ntask 크기만한 부분립방체를 얻으며 이 모든 마디들에 프로그람과제를 적재한다. iPSC/860의 조작체계는 매개 사용자들에게 10개까지의 부분립방체들을 배정하므로 iPSC/860에서는 pvm_spawn()을 한번 호출하여 과제들을 실행시키는것이 좋다.

iPSC/860에서 제각기 실행된 두조의 과제블로크들은 비록 그것들이 서로 다른 부분립방체에 있다 하더라도 다른 병렬가상기계과제들과 마찬가지로 서로 통신할수 있다. iPSC/860조작체계는 마디들로부터 체계외부에로 나가는 통보를 256kbyte아래로 제한하고있다.

```
int info = pvm_kill( int tid )
call pvmfkill( tid, info )
```

pvm_kill()함수는 TID로 지정된 병렬가상기계과제들을 중지한다. 이 함수는 kill지령을 호출한 과제는 중지하지 않으며 호출한 과제는 pvm_exit()를 리용하여 중지한다.

```
int info = pvm_catchout( FILE *ff )
call pvmfcatchout( onoff )
```

병렬가상기계는 실행하는 과제들의 표준출구와 표준오유출구를 /tmp/pvml.<uid>에 기록하도록 기정으로 지정하고있다. pvm_catchout()함수는 생성된 모든 자식과제들의 표준출구를 어미과제에게 돌려주는 기능을 수행한다.

pvmd는 자식과제들이 자기의 표준출구나 표준오유출구에 출구하는 문자들을 수집하여 어미과제에 조종통보로 보내주며 어미과제는 매 행마다 표식을 달아 전용파일에 그것을 첨부한다.

출구집합이 가능할 때 어미과제가 pvm_exit()함수를 호출하면 그는 표준출구를 내보내는 모든 과제들이 자기의 결과를 내보내고 중지할 때까지 기다린다.

이것을 피하기 위하여 프로그람작성자는 pvm_exit()함수를 호출하기전에 pvm_catchout(0)을 호출하여 모든 출구집합을 완료한다.

병렬가상기계는 새로운 처리기들을 추가하고 과제들을 처리기에 제출하며 새로운 과제들을 실행하는 처리를 위하여 특별한 병렬가상기계과제들을 등록하는 기능을 지원하고있다.

이것은 병렬가상기계를 리용하여 병렬가상기계일감들을 묶음방식으로 처리하는 개선된 묶음처리계획화를 위한 대면부로써 설계되였다(그러한 실례로 Condor, DQS, LSF 등을 들수 있다.).

이 등록함수들은 또한 병렬가상기계를 위한 오유수정도구들을 개발하기 위하여 오유제거기작성자들을 위한 대면부로서 리용할수 있다.

이 함수이름들은 pvm_reg_rm(), pvm_reg_hoster() 그리고 pvm_reg_tasker()이다.

2.2 정보

```
int tid = pvm_parent( void )
call pvmfparent( tid )
```

pvm_parent() 함수는 이 과제를 실행시킨 어미과제의 TID 혹은 이 과제가 pvm_spawn()에 의해 창조된것이 아니라면 PvmNoParent 값을 돌려준다.

```
int dtid = pvm_tidtohost( int tid )
call pvmftidtohost( tid, dtid )
```

pvm_tidtohost()함수는 같은 처리기에서 함께 실행되는 데몬의 TID를 반환한다. 이 함수는 이 과제가 어느 처리기에서 실행되는가를 결정할 때 편리하다. 구성된 처리기들의 이름과 같이 가상기계전체에 대한 보다 일반적인 정보는 다음의 기능을 리용하여 얻을수 있다.

```
int info = pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp )
call pvmfconfig( nhost, narch, dtid, name, arch, speed, info)
```

pvm_config()함수는 처리기들의 개수 nhost, 여러가지 자료형식들의 개수 narch등 가상기계에 대한 여러가지 정보를 돌려준다.

hostp는 리용자가 정의한 pvmhostinfo구조배렬에 대한 지적자이다. 배렬은 적어도 nhost 크기여야 한다.

매 pvmhostinfo구조체는 pvmd의 TID, 처리기이름, 구성 방식이름, 그 처리기의 상대적인 CPU속도와 같은 정보들을 포함한다.

포트란함수는 한번 호출할 때 한개의 처리기에 대한 정보를 돌려준다. 따라서 pvmfconfig호출을 nhost번 하면 전체 가상기계에 대한 정보가 얻어진다.

현재는 pvmfconfig()호출을 재설정하는 방법이 없으며 도중에 그것을 재시동하게 되여있다.

int info = pvm_tasks(int which, int *ntask, struct pvmtaskinfo **taskp)
call pvmftasks(which, ntask, tid, ptid, dtid, flag, aout, info)

pvm_tasks()함수는 가상기계에서 실행하는 병렬가상기계과제들에 대한 정보를 돌려준다.

현재 추가선택은 0을 포함하고있는데 이것은 모든 과제들과 pvmd를 의미한다. 과제들의 개수는 ntask에 들어있다.

taskp는 pvmtaskinfo구조배렬에 대한 지적자이다. 배렬의 크기는 ntask이다. 매개 pvmtaskinfo구조체는 TID, pvmd의 TID, 어미과제의 TID, 상태기발 그리고 실행하는 과제의 이름을 포함하고있다(병렬가상기계는 수동적으로 실행된 과제들의 파일이름은 모른다. 그러므로 이 경우에는 공백으로 남겨둔다.).

포트란함수는 한개의 호출에서 한개의 과제에 대한 정보만 돌려주므로 where = 0인 경우에는 pvmftasks가 ntask회수만큼 호출해야 모든 과제들에 대한 정보를 얻을수 있다.

포트란실행에서는 이 처리가 실행되는 동안에는 과제렬의 상태가 변하지 않는다고 가정한다. 만일 렬이 변하면 변경된 상태들은 다음번 호출을 할 때까지 반영되지 않는다.

pvm_config()와 pvm_tasks()에 대한 실례는 병렬가상기계조작탁프로그람의 원천에서 직접 찾을수 있다.

2.3 동적구성

int info = pvm_addhosts(char **hosts, int nhost, int *infos)
int info = pvm_delhosts(char **hosts, int nhost, int *infos)
call pvmfaddhost(host, info)
call pvmfdelhost(host, info)

C함수들은 가상기계에 처리기들의 모임을 첨부하거나 삭제한다. 포트란함수들은 가상기계에 한개의 처리기를 추가하거나 삭제한다.

포트란 함수에서 info는 1 혹은 상태코드를 돌려준다.

C함수에서 info는 가상기계에 실제로 첨부된 처리기들의 개수를 돌려준다. 변수 infos는 가상기계에 추가되는 개별적인 처리기들에 대한 상태코드를 포함하고있는 길이가 nhost인 배렬이다. 이것을 리용하여 리용자는 처리기들의 모임 전체를 다시 추가하거나 삭제하지 않고 문제가 생긴 한개의 처리기만을 검사할수 있다.

이 함수들은 때때로 가상기계를 설정하는데 리용할수는 있지만 대체로는 큰 응용프로그람들에서 오유허용능력을 개선하기 위하여 리용한다.

이 함수들은 문제가 점점 풀기 어려워진다는것을 결정한 경우 응용프로그람 자체로 가상기계의 처리능력을 증가하는데 리용할수 있다.

그 한가지 실례는 CAD/CAM프로그람인데 여기서는 계산기간에 유한요소 격자를 세분화하면서 문제의 크기를 크게 증가시킨다. 또 한가지 사용법은 처리기의 오유를 검출하고 새로운 처리기로 교체함으로써 응용프로그람의 오유허용능력을 증가하는것이다.

2.4 신호기수법

```
int info = pvm_sendsig( int tid,  int signum )
call pvmfsendsig( tid,  signum,  info )
int info = pvm_notify( int what,  int msgtag,  int cnt, int tids )
call pvmfnotify( what,  msgtag,  cnt,  tids,  info )
```

pvm_sendsig()함수는 TID로 지정된 다른 병렬가상기계과제에 신호기 signum을 보낸다. pvm_notify는 호출자가 어떤 사건을 검출하였다는것을 병렬가상기계에게 통보한다.

추가선택들은 다음과 같다.

PvmTaskExit – 과제가 끝났음을 통보한다.
PvmHostDelete – 처리기가 삭제되였음을 통보한다.
PvmHostAdd – 처리기가 추가되였음을 통보한다.

통보요구에 응답하여 병렬가상기계는 몇개의 통보들을 호출한 과제에게 보낸다. 통보들은 리용자가 제공한 통보표적들을 붙인다.

tids배렬은 누가 언제 TaskExit나 HostDelete를 리용하는가를 알려준다. 이 배렬은 HostAdd를 리용할 때는 아무것도 포함하지 않는다.

만일 요구하면 pvm_config()함수와 pvm_tasks()함수는 과제와 pvm데몬의 tids를 얻는데 리용할수 있다.

만일 과제 A를 실행하는 처리기가 실패하고 과제 B가 과제 A가 완료하였는가를 문의하였다면 과제 B는 그것이 처리기의 실패로 인한 간접적인 완료라 할지라도 통보를 받는다.

2.5 추가선택설정과 얻기

```
int oldval = pvm_setopt( int what, int val )
int val = pvm_getopt( int what )
call pvmfsetopt( what, val, oldval )
call pvmfgetopt( what, val )
```

pvm_setopt()함수는 병렬가상기계체계에서 리용자가 추가선택들을 설정하거나 얻을수 있게 하는 일반적인 함수이다. 병렬가상기계에서 pvm_setopt()함수는 자동적인 오유통보인쇄, 오유제거준위설정, 모든 병렬가상기계호출들에 대한 통신경로지정을 비롯한 여러가지 추가선택들을 리용할수 있다.

pvm_setopt()함수는 oldval에 있는 이전값들을 돌려준다.

병렬가상기계에서 what는 다음의 값들을 가질수 있다.

추가선택	값	의미
PvmRoute	1	경로지정전략
PvmDebugMask	2	오유제거마스크
PvmAutoErr	3	자동오유보고
PvmOutputTid	4	자식처리에 대한 표준 출구목적지

PvmOutputCode	5	출구통보표적
PvmTraceTid	6	자식처리의 추적목적지
PvmTraceCode	7	추적통보표적
PvmFragSize	8	통보토막크기
PvmResvTids	9	예약된 표적들과 tid들을 통보에 리용
PvmSelfOutputTid	10	자체의 표준출구목적지
PvmSelfOutputCode	11	출구통보표적
PvmSelfTraceTid	12	자체의 표준출구목적지
PvmSelfTraceCode	13	추적통보표적

pvm_setopt()함수는 병렬가상기계과제들사이에 직접 통신을 지정하려고 할 때 아주 많이 리용된다. 일반적인 규칙으로는 pvm_setopt(PvmRoute, PvmRouteDirect)를 리용하여 망에서 병렬가상기계의 통신대역을 두배로 할수 있다. 약점은 이러한 빠른 통신방법을 Unix체계에서는 적용할수 없다는것이다.

그러므로 어떤 병렬가상기계응용프로그람이 호상 통신하는 60개이상의 과제들을 가지고있다면 동작하지 않는다.

만일 동작하지 않으면 병렬가상기계는 자동적으로 기정의 통신방법으로 돌아간다. 이것은 응용프로그람의 실행중에 여러번 호출되여 과제-과제사이에 직접 통신련결을 실현할수 있다.

그러나 일반적으로는 pvm_mytid()함수를 호출한 다음에 한번만 사용한다.

2.6 통보전달

병렬가상기계에서 통보전송은 세 단계로 진행한다.

첫째: pvm_initsend()함수나 pvm_mkbuf()함수를 호출하여 송신완충기를 초기화한다.

둘째: 몇개의 pvm_pk*()함수를 사용하여 통보들을 이 완충기에 압축한다(포트란에서 모든 통보압축은 pvmfpack()함수를 리용하여 진행한다.).

셋째: pvm_send()함수로 통보를 다른 프로쎄스에 보낼수 있으며 pvm_mcast()함수를 통하여 다중통신할수 있다. 봉쇄 혹은 비봉쇄수신함수를 호출하여 통보를 수신하며 수신완충기에 있는 압축된 항목들을 푼다.

수신함수들은 임의의 통보 혹은 지적된 원천으로부터 임의의 통보를 접수할수 있으며 어떤 통보표적이 붙은 통보 혹은 주어진 원천으로부터 특정의 통보표적이 붙은 통보를 접수할수 있다.

또한 통보가 접수되지는 않았지만 도착하였는지 검사하는 문의함수도 있다. 만일 요구한다면 다른 수신문맥들도 병렬가상기계에서 조종할수 있다.

pvm_recvf()함수를 리용하여 리용자들은 병렬가상기계수신함수들에서 리용할 자기자체의 수신문맥들을 정의할수 있다.

2.6.1 통보완충기들

```
int bufid = pvm_initsend( int encoding )
call pvmfinitsend( encoding,  bufid )
```

만일 리용자가 한개의 송신완충기를 가지고있다면(대체로 이것은 일반적인 경우이다.) pvm_initsend()함수만을 리용한다. 이것은 새로운 통보를 완충기에 압축하기전에 호출된다.

pvm_initsend()함수는 송신완충기를 지우고 새로운 통보를 압축하기 위하여 한개의 완충기를 창조한다. 새로운 완충기식별자는 bufid에 보관된다.

부호화추가선택들은 다음과 같다.

PvmDataDefault - XDR 부호화는 이 통보를 보내기전에 리용자가 이종의 처리기를 체계에 첨부하였는지 모르기때문에 기정으로 진행한다. 만일 리용자가 그의 자료처리형식을 알고있는 처리기에 통보를 보낸다면 PvmDataRaw 부호화방법을 리용한다.

PvmDataRaw – 부호화를 진행하지 않는다. 통보들은 자기 본래의 형식으로 전송된다. 만일 수신하는 프로쎄스가 이 형식을 리해하지 못하면 통보풀기시에 오유를 내보낸다.

PvmDataInPlace – 자료는 압축을 위하여 어떤 장소에 보관된다. 완충기들은 다만 전송해야 할 항목들의 크기와 지적자들만을 가지고있다.

pvm_send()함수를 호출하면 항목들은 즉시 체계구역에 보관된다. 이 추가선택은 리용자가 통보들을 압축하고 또 그것들을 전송하는 사이에 항목들을 수정하지 않아도 되므로 통보를 복사하는 회수를 줄일수 있다.

이것은 한번 압축한 다음에는 수정하여 응용프로그람의 실행시에 여러번 전송할수 있다.

만일 리용자가 응용프로그람내에서 여러개의 통보완충기들을 관리하려고 한다면 다음의 통보완충기함수들을 리용한다.

대다수의 프로쎄스들사이통신에서는 다중통보완충기들이 필요없다.

병렬가상기계에서는 어떤 시점에서 매 프로쎄스당 한개의 능동인 송신완충기와 한개의 능동인 수신완충기가 있다.

개발자는 여러개의 통보완충기들을 창조하고 그것들을 서로 절환하면서 자료의 압축과 전송을 실현할수 있다. 압축, 송신, 수신, 풀기는 오직 능동완충기들에 대해서만 효력을 가진다.

```
int bufid = pvm_mkbuf( int encoding )
call pvmfmkbuf( encoding, bufid )
```

pvm_mkbuf()함수는 새로운 빈 송신완충기를 창조하며 통보압축을 위해 리용하는 부호화방법을 지정한다. 이 함수는 완충기식별자 bufid를 돌려준다.

```
int info = pvm_freebuf( int bufid )
call pvmffreebuf( bufid, info )
```

pvm_freebuf()함수는 식별자가 bufid인 완충기를 해제한다. 이것은 통보를 보낸 다음 더 이상 완충기가 필요없을 때 진행한다. 또 필요하면 pvm_mkbuf()함수를 호출하여 새로운 통보완충기를 창조한다.

pvm_initsend()함수를 리용할 때는 이 함수들중 어느것도 리용할수 없다. 왜냐하면 이 함수들은 리용자들을 위한 함수이기때문이다.

```
int bufid = pvm_getsbuf( void )
call pvmfgetsbuf( bufid )
int bufid = pvm_getrbuf( void )
call pvmfgetrbuf( bufid )
```

pvm_getsbuf()는 능동인 송신완충기식별자를 돌려준다.
pvm_getrbuf()는 능동인 수신완충기식별자를 돌려준다.

```
int oldbuf = pvm_setsbuf( int bufid )
call pvmfsetrbuf( bufid, oldbuf )
int oldbuf = pvm_setrbuf( int bufid )
call pvmfsetrbuf( bufid, oldbuf )
```

이 함수들은 능동인 송신(수신)완충기들을 bufid에 설정하고 이전 완충기의 상태를 보관하고 이전의 능동완충기식별자를 oldbuf에 설정한다.

만일 pvm_setsbuf()나 pvm_setrbuf()에서 bufid를 0으로 설정하면 현재의 완충기가 보관되며 능동완충기는 존재하지 않는다. 이 특징을 리용하여 응용프로그람통보들의 현재 상태를 보관함으로써 역시 병렬가상기계통보들을 리용하는 수학서고나 그라픽스대면부들이 응용프로그람통보들과 서로 간섭하지 않게 할수 있다. 그것들을 완료하면 응용프로그람의 완충기들을 능동으로 재설정할수 있다.

리용자는 통보완충기함수들을 리용하여 통보들을 다시 압축하지 않고도 전송할수 있다. 이에 대해서는 다음에 설명한다.

```
bufid = pvm_recv( src, tag );
oldid = pvm_setsbuf( bufid );
info = pvm_send( dst, tag );
```

```
info = pvm_freebuf( oldid );
```

2.6.2 자료의 압축

다음의 C함수들은 주어진 자료형을 가진 배렬들을 능동인 송신완충기에 압축한다. 이 함수들은 자료를 한개의 통보에 압축하기 위해 여러번 호출된다. 그러므로 한개의 통보는 서로 다른 자료형을 가진 여러개의 배렬들을 가지고있을수 있다. C구조체들은 그의 매 요소들을 압축해야 한다.

통보들은 임의로 압축할수 있지만 응용프로그람들은 통보들을 자기가 압축한대로 풀어야 한다. 이것은 엄밀하게 요구되지는 않지만 프로그람 작성에서는 이 규칙을 지켜야 한다.

함수에서 매개 변수들은 압축해야 할 첫 항목에로의 지적자이다. nitem은 이 배렬에서 압축해야 할 항목들의 전체 개수이며 stride는 압축할 때 리용하는 걸음수이다.

한가지 례외적인것은 pvm_pkstr()인데 이것은 문자렬들이 NULL문자로 끝나도록 정의하였기때문에 nitem과 stride변수가 필요없다.

```
int info = pvm_pkbyte( char *cp,  int nitem,  int stride )
int info = pvm_pkcplx( float *xp,  int nitem,  int stride )
int info = pvm_pkdcplx( double *zp,  int nitem, int stride )
int info = pvm_pkdouble( double *dp,  int nitem, int stride )
int info = pvm_pkfloat( float *fp,  int nitem,  int stride )
int info = pvm_pkint( int *np,  int nitem,  int stride )
int info = pvm_pklong( long *np,  int nitem,  int stride )
int info = pvm_pkshort( short *np,  int nitem,  int stride )
int info = pvm_pkstr( char *cp )
int info = pvm_packf( const char *fmt,  ... )
```

병렬가상기계는 또한 무슨 자료를 압축하며 그것을 송신완충기에 어떻게 압축하는가를 지적하는 C언어에서의 인쇄지령과 류사한 형식의 표기법을 리용하는 압축함수들을 제공하고있다.

count와 stride를 지적하면 모든 변수들은 주소로써 넘어가며 다른 경우 변수들은 값으로 취급된다.

포트란에서는 한개의 함수가 우에서 서술한 C함수들의 기능을 담당한다.

```
call pvmfpack( what, xp, nitem, stride, info )
```

변수 xp는 압축해야 할 배렬의 첫 항목이다. 포트란에서는 압축해야 할 문자렬에 있는 문자의 개수를 nitem으로 설정한다.

옹근수 what는 압축해야 할 자료의 형을 지적한다.

추가선택들은 다음과 같다.

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

이 이름들은 pvm/include/fpvm.h에 정의되여있다.

2.6.3 자료의 송신과 수신

```
int info = pvm_send( int tid, int msgtag )
call pvmfsend( tid, msgtag, info )
int info = pvm_mcast( int *tids, int ntask, int msgtag )
call pvmfmcast( ntask, tids, msgtag, info )
```

pvm_send()함수는 통보들에 옹근수값을 가진 통보표적을 붙여 프로쎄스 TID에 직접 보낸다.

pvm_mcast()함수는 통보들에 옹근수값을 가진 통보표적을 붙여 옹근수배렬 tids(자기는 제외하고)에 지적된 모든 과제들에 전송한다.

tids배렬의 길이는 ntask이다.

int info = pvm_psend(int tid, int msgtag, void *vp, int cnt, int type)

call pvmfpsend(tid, msgtag, xp, cnt, type, info)

pvm_psend()함수는 지적된 자료형을 가진 배렬을 압축하고 TID로 지적된 과제에 보낸다. 포트란에서 정의된 자료형들은 pvmfpack()에서와 같다.

C에서 형변수들은 다음과 같다.

PVM_STR	PVM_FLOAT
PVM_BYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE
PVM_INT	PVM_DCPLX
PVM_LONG	PVM_UINT
PVM_USHORT	PVM_ULONG

병렬가상기계에는 여러가지 형태의 통보전송방법들이 있다.

병렬가상기계에서는 송수신명령들이 서로 일치해야 한다. 실례로 pvm_psend는 pvm_precv와 일치하여야 한다.

다음 함수들은 어느것이나 다 통보가 어떻게 발송되였는가에 관계없이 임의의 들어오는 통보에 대하여 호출된다.

int bufid = pvm_recv(int tid, int msgtag)

call pvmfrecv(tid, msgtag, bufid)

이 봉쇄수신함수는 msgtag를 가진 통보가 TID로부터 도착할 때까지 기다린다. 다음에 통보를 창조된 새로운 능동수신완충기에 배치한다.

이전의 능동수신완충기는 그것이 pvm_setrbuf()호출로 보관하지 않는 한 지워진다.

int bufid = pvm_nrecv(int tid, int msgtag)

call pvmfnrecv(tid, msgtag, bufid)

만일 요구된 통보가 도착하지 않으면 이때 비봉쇄수신함수 pvm_nrecv()는 bufid=0을 되돌린다. 이 함수를 같은 통보에 대하여 여러번 호출하여 통보가 도착하였는가를 검사하면서 그사이에 다른 처리를 진행할수 있다.

다른 처리를 진행하지 않을 때 봉쇄수신함수 pvm_recv()를 호출하여 통보의 도착을 검사할수 있다.

msgtag표식을 가진 통보가 TID로부터 도착하였다면 pvm_nrecv()는 이 통보를 새로운 (창조된)능통통보완충기에 배치하며 이 완충기의 id를 돌려준다. 이전의 능동수신완충기는 그것이 pvm_setrbuf()함수호출로 보관하지 않는 한 지워진다.

```
int bufid = pvm_probe( int tid,  int msgtag )
call pvmfprobe( tid,  msgtag,  bufid )
```

만일 요구하는 통보가 도착하지 않으면 이때 비봉쇄수신함수 pvm_probe()는 bufid=0을 되돌린다. 다른 경우 통보의 bufid를 돌리지만 그것을 접수하지 않는다.

이 함수를 같은 통보에 대하여 여러번 호출하여 통보가 도착하였는가를 검사하면서 그 사이에 다른 처리를 진행할수 있다. 추가적으로 귀환된 bufid를 가지고 pvm_bufinfo()를 호출하여 통보를 접수하기전에 그에 대한 정보를 결정한다.

```
int bufid = pvm_trecv( int tid,  int msgtag, struct timeval *tmout )
call pvmftrecv( tid,  msgtag,  sec,  usec, bufid )
```

병렬가상기계는 또한 수신에서 시간초과기능을 지원하고있다.

오유나 실패로 하여 통보가 절대로 도착하지 않는 경우를 고찰하자.

pvm_recv()함수는 영원히 봉쇄될것이다.

이러한 경우를 피하기 위하여 리용자는 어떤 지정된 시간마큼 기다린 다음에는 그것을 포기하려고 한다.

pvm_trecv()함수는 사용자가 기다림시간량을 지정하게 한다. 만일 시간주기가 대단히 크면 pvm_trecv는 pvm_recv처럼 동작한다. 시간주기를 령으로 설정하면 pvm_trecv는 pvm_nrecv처럼 동작한다. 따라서 pvm_trecv는 봉쇄수신과 비봉쇄수신 기능들사이의 공백을 없앤다.

```
int info = pvm_bufinfo( int bufid,  int *bytes, int *msgtag, int *tid )
call pvmfbufinfo( bufid,  bytes,  msgtag,  tid,  info )
```

pvm_bufinfo()함수는 bufid로 지적된 통보의 msgtag, 원천 TID, 바이트단위로 지정된 길이를 돌려준다. 이것은 임의로 접수된 통보의 표식과 원천을 결정하는데 리용할수 있다.

```
int info = pvm_precv( int tid,  int msgtag,  void *vp,
        int cnt,  int type,  int *rtid,  int *rtag,  int *rcnt )
call pvmfprecv( tid, msgtag, xp, cnt, type, rtid, rtag, rcnt, info )
```

pvm_precv()함수는 봉쇄수신기능과 수신된 완충기내용의 풀기조작을 동시에 수행한다. 이것은 bufid를 돌려주지 않는다. 대신에 TID, msgtag, 그리고 cnt의 실제적인 값들을 돌려준다.

```
int (*old)() = pvm_recvf(int (*new)(int bufid, int tid, int tag))
```

pvm_recvf()함수는 수신함수들에서 리용되는 수신문맥을 수정하고 병렬가상기계를 확장하는데 리용된다. 기정의 수신문맥은 원천과 통보표적을 정합하는것이다. 이것은 리용자가 정의한 비교함수에 의해 수정될수 있다.

pvm_recvf()에 대한 포트란대면부함수는 없다.

2.6.4 자료의 풀기

다음의 C함수들은 능동인 수신완충기로부터 자료형들을 풀어낸다.

응용프로그람에서 대응하는 풀기함수들은 형과항목들의 개수, stride가 일치하여야 한다. nitem은 풀어야 할 항목들의 개수이다. stride는 걸음이다.

```
int info = pvm_upkbyte( char *cp,  int nitem,  int stride )
int info = pvm_upkcplx( float *xp,  int nitem,  int stride )
int info = pvm_upkdcplx( double *zp,  int nitem, int stride )
int info = pvm_upkdouble( double *dp,  int nitem, int stride )
int info = pvm_upkfloat( float *fp,  int nitem,  int stride )
int info = pvm_upkint( int *np,  int nitem,  int stride )
int info = pvm_upklong( long *np,  int nitem,  int stride )
int info = pvm_upkshort( short *np,  int nitem, int stride )
int info = pvm_upkstr( char *cp )
int info = pvm_unpackf( const char *fmt,  ... )
```

pvm_unpackf()는 인쇄지령과 류사한 형식의 표기법을 리용하여 무슨 자료를 풀기하며 그것을 수신완충기에서 어떻게 풀기하는가를 지적한다. 포트란함수는 한개의 지령으로 우의 모든 기능들을 서술할수 있다.

```
call pvmfunpack( what,  xp,  nitem,  stride,  info )
```

변수 xp는 통보를 풀어야 할 배렬을 지적한다. 옹근수변수 what는 풀어야 할 자료의 형을 지적한다.

2.7 동적인 프로쎄스그룹들

동적인 프로쎄스그룹기능들은 핵심기능을 수행하는 병렬가상기계함수들의 우에 구축되였다. 리용자는 병렬가상기계서고 libgpvm.a를 리용자프로그람과 결합하여 그룹기능들을 리용할수 있다.

pvmd는 그룹기능들을 수행하지 않는다. 이 과제는 그룹봉사기에 의해 조종되며 첫번째 그룹기능이 실행될 때 자동적으로 실행된다.

통보전송대면부에서 그룹들을 어떻게 조종하겠는가에 대한 문제에서 일련의 문제들이 제기된다. 기본문제는 효과성과 믿음성이며 정적 대 동적그룹들사이의 절충안이다. 병렬가상기계에서 그룹기능들은 리용자에게 대단히 일반적이며

효과적으로 리용하도록 설계되였다. 어떤 병렬가상기계과제는 임의의 시각에 같은 그룹에 있는 다른 과제들에 통보하지 않고 그룹에 결합하거나 그룹에서 탈퇴할수 있다.

과제들은 다른 그룹에 속한 과제들에 통보를 보낼수 있다. 일반적으로 병렬가상기계과제는 임의의 시점에서 다음의 그룹기능들을 호출할수 있다.

례외로 되는것들은 pvm_lvgroup(), pvm_barrier() 그리고 pvm_reduce() 인데 이 지령들은 그 자체의 특성상 호출하는 과제가 특수한 그룹의 성원일것을 요구한다.

```
int inum = pvm_joingroup( char *group )
int info = pvm_lvgroup( char *group )
call pvmfjoingroup( group, inum )
call pvmflvgroup( group, info )
```

이 함수들은 리용자가 정의한 그룹에 과제를 결합하거나 내보낸다.

첫 pvm_joingroup()를 호출하여 group라는 이름을 가진 그룹을 창조하며 호출한 과제를 이 그룹에 넣는다.

pvm_joingroup()는 이 그룹에 있는 프로쎄스의 실체번호 (inum)를 돌려준다. 실체번호들은 0부터 그룹성원들의 개수이다.

병렬가상기계에서 과제는 여러 그룹에 결합할수 있다. 만일 프로쎄스가 그룹을 리탈하였다가 다시 결합한다면 그 과제는 다른 실체번호를 가지게 된다.

실체번호들은 그룹에 결합하는 과제들이 될수록 제일 작은 번호를 가지도록 되여있다. 그러나 많은 과제들이 한개의 그룹에 결합된다면 과제가 이전의 실체번호를 받는다는 담보는 없다.

```
int tid = pvm_gettid( char *group, int inum )
int inum = pvm_getinst( char *group, int tid )
int size = pvm_gsize( char *group )
call pvmfgettid( group, inum, tid )
call pvmfgetinst( group, tid, inum )
```

call pvmfgsize(group, size)

pvm_gettid()는 주어진 그룹이름과 실체번호를 가진 프로세스의 TID를 돌려준다. pvm_gettid()는 두개의 과제들이 호상 다른 과제들에 대하여 몰라도 같은 그룹에 결합하기만 하면 다른 과제의 TID를 알수 있게 한다.

pvm_getinst()함수는 지적된 그룹에 있는 TID의 실체번호를 돌려준다.

pvm_gsize()함수는 지적된 그룹의 성원개수를 돌려준다.

int info = pvm_barrier(char *group, int count)

call pvmfbarrier(group, count, info)

pvm_barrier()를 호출한 프로세스는 그룹에 있는 count 개수의 성원들이 pvm_barrier를 호출할 때까지 봉쇄된다. 일반적으로 count는 그룹에 있는 전체 성원들의 개수이다.

동적인 프로쎄스그룹을 가진 병렬가상기계는 그룹에 몇개의 성원들이 있는지 모르기때문에 count가 필요하다. 그룹의 성원이 아닌 프로쎄스가 그 그룹에 대하여 pvm_barrier을 호출하면 오유가 발생한다. 또한 호출할 때 넘겨지는 변수들이 일치하지 않을 때도 오유가 생긴다.

실례로 그룹의 어떤 성원이 count=4를 가지고 pvm_barrier를 호출하고 다른 성원이 count=5를 가지고 pvm_barrier를 호출하면 오유가 생긴다.

int info = pvm_bcast(char *group, int msgtag)

call pvmfbcast(group, msgtag, info)

pvm_bcast()는 통보에 옹근수식별자 msgtag를 붙이며 이 통보를 자기를 제외한 그룹내의 모든 과제들에 방송한다. pvm_bcast()에서 모든 과제들이라는것은 이 함수를 호출할 때 그룹봉사기가 인식하고있는 그룹내의 모든 과제들을 의미한다.

만일 과제들이 통보를 방송하는 기간에 그룹에 결합된다면 그 과제들은 이 통보를 받지 못할것이다. 또 과제들이 통보를 방송할 때 이 그룹을 떠난다면 통보의 복사물이 그 과제에 보내질것이다.

```
int info = pvm_reduce( void (*func)(), void *data, int nitem, int
                        datatype, int msgtag, char *group, int root )
call pvmfreduce( func, data, count, datatype, msgtag, group, root,
                 info )
```

pvm_reduce()는 그룹전반에서 대역적인 산수연산들, 실례로 대역더하기와 대역덜기를 진행한다. 연산의 결과는 뿌리프로쎄스에 나타난다.

병렬가상기계는 리용자가 func로 서술할수 있는 4개의 함수를 지원하고있다. 그것들은 다음과 같다.

PvmMax

PvmMin

PvmSum

PvmProduct

이 연산은 입구자료에 대해서 요소별로 진행한다. 실례로 만일 자료배렬이 두개의 류점연산수들이고 func는 PvmMax라면 이 계산결과는 두개의 수를 포함하고있다. 즉 제일 큰값과 그룹내의 성원들중에서 제일 큰값을 가지고있는 성원을 포함하고있다.

보충적으로 리용자는 자기 자체의 대역연산조작을 정의할수도 있다.

그 실례는 병렬가상기계원천코드에서 찾을수 있다. 상세한 정보들은 $PVM_ROOT/examples/gexample.c를 참조하면 된다.

주의: pvm_reduce()는 봉쇄되지 않는다.

만일 과제가 pvm_reduce를 호출하고 결과를 받기전에 그룹을 떠난다면 오유가 발생한다.

3. 프로그람실례

여기서는 병렬가상기계프로그람들을 구체적으로 설명한다.

첫번째 실례프로그람인 fork-join.c는 프로세스들을 생성하고 동기화하는 방법에 대하여 서술한다. 두번째 실례로서는 포트란언어로 서술한 스칼라적 계산프로그람 PSDOT를 서술한다.

세번째 실례프로그람 failure.c는 리용자가 pvm_notify()를 리용하여 오유허용능력을 가진 응용프로그람을 개발할수 있는 방법을 서술한다. 여기서는 행렬곱하기프로그람을 제시하였다.

마지막으로 병렬가상기계를 리용하여 도선에서의 열확산방정식을 계산하는 실례프로그람을 주었다.

3.1 fork-join

이 프로그람에서는 병렬가상기계과제들을 생성하고 그들사이의 동기를 보장하는 방법에 대하여 서술한다. 프로그람은 여러개의 과제들을 생성한다.

기정으로 세개의 과제를 생성한다. 자식프로쎄스들은 다음에 어미과제에 통보를 보냄으로써 동기를 실현한다. 어미는 생성된 매 과제들로부터 통보를 접수하고 그 통보의 내용을 화면에 출력한다. fork-join프로그람은 어미과제와 자식과제들에 대하여 각각 코드를 작성하였다.

좀 더 구체적으로 실험해보자.

프로그람에서는 제일 먼저 pvm_mytid()를 호출한다. 이 기능은 다른 병렬가상기계기능들을 호출하기전에 먼저 호출되여야 한다.

pvm_mytid()호출의 결과는 항상 정의옹근수이다. 정의옹근수가 아니라면 오유가 발생한것으로 된다.

fork-join프로그람에서는 이 기능을 호출하여 mytid의 값을 검사하며 만일 오유가 발생하면 pvm_error()를 호출하여 프로그람을 끝낸다.

pvm_perror()는 마지막 병렬가상기계호출에서 무엇이 잘못되였는가를 표시하는 통보를 출력한다. 실례프로그람에서 마지막 호출은 pvm_mytid()이므로 pvm_perror()는 병렬가상기계가 이 처리기에서 시동되지 않았다는것을 알리는 통보를 내보낸다.

pvm_perror()에서 변수는 그 어떤 오유통보에 pvm_perror()함수가 덧붙인 어떤 문자렬이다. 우리의 경우에는 argv[0]을 넘기는데 이것은 지령행에서 입력한 프로그람의 이름이다. pvm_perror()는 Unix체계의 perror()함수를 리용하여 작성하였다.

유효한 mytid를 얻었다고 가정하고 pvm_parent()를 호출한다.

pvm_parent()는 이 과제를 생성한 과제(어미과제)의 과제 TID를 돌려준다. 초기의 fork-join프로그람은 Unix체계의 쉘에서 시작하였으므로 이 초기과제는 어미를 가지지 않는다. 이 과제는 그 어떤 다른 병렬가상기계과제에 의해서가 아니라 리용자가 수동적으로 실행시킨것이다.

초기의 fork-join과제에 대하여 pvm_parent()는 어떤 과제의 id를 돌려주지 않으며 오유코드 PvmNoParent를 내보낸다.

따라서 프로그람에서는 pvm_parent()호출의 결과가 PvmNoParent인가 아닌가를 검사하는 방법으로 어미과제인가 혹은 자식과제인가를 가를수 있다. 만일 이 과제가 어미라면 자식과제들을 생성하며 어미가 아니라면 어미에게 통보를 보낸다.

어미과제가 실행한 코드를 검사해보자. 과제들의 개수는 지령행에서 argv[1]로 주어진다. 과제의 개수가 적당치 않으면 pvm_exit()를 호출하여 프로그람을 끝내고 귀환한다.

pvm_exit()는 병렬가상기계에게 이 프로그람이 더는 병렬가상기계의 기능들을 리용하지 않는다는것을 통보하므로 대단히 중요하다. (이 경우에 과제는 완료하며 병렬가상기계는 이 과제가 더 이상 자기의 봉사를 받지 않는다는것을 알게 된다.)

과제들의 개수가 적당하다고 인정하면 fork-join은 다음에 자식과제들을 생성하려고 시도한다. pvm_spawn()은 병렬가상기계에게 argv[0]에 있는 이름을 가진 ntask개의 과제들을 창조할것을 요구한다. 두번째 파라메터는 창조된 과제에게 넘기는 변수목록이다. 이 경우에 우리는 자식과제들에 특별한 변수들을 넘길 필요가 없으므로 이 값은 null이다.

세번째 변수인 PvmTaskDefault는 병렬가상기계에게 임의의 처리기에서 과제들을 실행할것을 요구하는 변수이다. 만일 리용자가 과제를 특별한 처리기

혹은 어떤 구성방식을 가진 처리기에서 실행시키려고 하는 경우에는 이 구역에 PvmTaskHost 혹은 PvmTaskArch를 입력하여 지정할수 있다.

우리는 과제를 어디에서 실행시키는가는 관계하지 않으므로 flag에 PvmTaskDefault를 지정하며 4개의 변수들을 null로 한다. 마지막으로 ntask는 몇개의 과제들을 실행하는가를 지적한다.

옹근수배렬 child는 새롭게 생성된 자식과제들의 과제id들을 보관하고있다.

pvm_spawn()의 귀환값은 실제로 생성된 과제들의 개수이다.

만일 info가 ntask와 일치하지 않으면 생성시에 어떤 오유가 발생한것으로 된다.

오유가 발생한 경우 과제의 id배렬의 child에 실제적인 과제의 id가 아니라 오유코드가 배치된다.

fork-join프로그람은 이 배렬들을 순환하면서 과제의 id 혹은 오유코드들을 출구한다.

만일 과제가 성과적으로 생성하였다면 프로그람은 완료한다.

매개 자식과제들에 대하여 어미과제는 통보를 접수하고 그 통보에 대한 정보를 출구한다. pvm_recv()호출은 임의의 과제로부터 통보를 접수한다.

pvm_recv()의 귀환값은 통보완충기를 표시하는 옹근수이다.

pvm_bufinfo()를 호출한다. 이것은 buf로 표시된 통보에 대하여 수신자 프로쎄스의 통보길이, 표적, 과제id를 얻는다.

fork-join에서 자식프로쎄스가 보낸 통보들은 단일한 옹근수값과 자식과제의 과제id를 가지고있다.

pvm_upkint()호출은 통보로부터 옹근수를 꺼내여 mydata변수에 보관한다. fork-join은 간단한 검사를 통하여 mydata의 값과 pvm_bufinfo()의 값을 검사한다. 만일 값이 다르면 프로그람은 오유가 있다고 보고 오유통보를 내보낸다. 마지막으로 통보에 대한 정보를 출구하고 어미프로그람을 끝낸다.

fork-join에서 마지막 코드토막은 자식과제가 실행한다. 통보완충기에 자료를 배치하기전에 완충기는 pvm_initsend()를 호출하여 초기화된다. PvmDataDefault변수는 자료가 목적처리기에 정확한 형식으로 도착한다는것을 담보하기 위하여 병렬가상기계가 어떤 자료변환을 진행하는가를 알려준다. 어떤 경우에는 이것이 불필요한 자료변환으로 될수 있다.

만일 목적처리기가 같은 자료형식을 리용하기때문에 자료변환이 필요없다면 PvmDataRaw를 pvm_initsend()의 변수로 리용할수 있다.

pvm_pkint()호출은 단일한 옹근수인 mytid를 통보완충기에 배치한다. 대응하는 통보풀기호출을 압축호출과 정확히 일치시키는것이 중요하다. 압축과 풀기는 일대일 대응되여야 한다. 마지막으로 통보는 통보표적을 리용하여 어미과제에 보내진다.

fork-join 실례프로그람

```
/*
fork-join 실례
프로쎄스들을 생성하고 통보들을 교환하는 방법들을 서술한다.
*/
/* 병렬가상기계 서고들을 위한 정의와 원형들 */
#include <stdio.h>
#include <pvm.h>
/* 프로그람에서 생성할 자식과제들의 최대개수 */
#define MAXNCHILD 20  /* 통보에서 사용하는 표적 */
#define JOINTAG 11
main(int argc, char *argv[])
{
 int ntask = 3; /* 생성할 과제들의 개수, 기정으로는 3 */
 int info;    /* pvm호출로부터의 귀환코드 */
 int mytid;  /* 이 프로그람의 과제id */
 int myparent; /* 어미과제의 id */
 int child[MAXNCHILD]; /* 배렬에 있는 자식과제 id */
 int i, mydata, buf, len, tag, tid;

mytid = pvm_mytid();  /* 이 프로그람의 과제id를 찾는다. */
if (mytid < 0)  { /* 오유검사  */
```

```
        pvm_perror(argv[0]); /* 오유출구 */
         return -1;      /* 프로그람완료 */
 }
myparent = pvm_parent();  /* 어미과제의 과제 id를 찾는다. */

/* PvmNoParent 가 아닌 오유가 발생하면 프로그람을 끝낸다. */
if ((myparent < 0) && (myparent != PvmNoParent)) {
       pvm_perror(argv[0]);
       pvm_exit();
       return -1;
}
/* 어미를 가지지 않는다면 내가 어미이다. */
if (myparent == PvmNoParent) {
    /* 생성해야 할 과제의 개수를 결정한다. */
   if (argc == 2) ntask = atoi(argv[1]);
   /* ntask가 의미를 가지도록 설정한다. */
     if ((ntask < 1) || (ntask > MAXNCHILD)) {
           pvm_exit();
           return 0;
     }

     /* 자식과제들을 생성한다. */
    info = pvm_spawn(argv[0], (char **)0, PvmTaskDefault, (char *)0,
                           ntask, child);
    /* 과제id들을 출구한다. */
   for (i = 0; i < ntask; i++)
      if (child[i] < 0)   /* 오유코드들을 10진수로 출구한다. */
          printf(" %d", child[i]);
      else       /* 과제id를 16진수로 출구한다. */
         printf("t%x\t", child[i]);
```

```
    putchar('\n');
     /* 생성이 성과적으로 진행되도록 한다. */
     if (info == 0) {
            pvm_exit();
            return -1;
     }
  ntask = info;
 for (i = 0; i < ntask; i++) {
        /* 임의의 자식처리로부터 통보를 접수한다. */
     buf = pvm_recv(-1, JOINTAG);
     if (buf < 0)
        pvm_perror("calling recv");
     info = pvm_bufinfo(buf, &len, &tag, &tid);
     if (info < 0)
        pvm_perror("calling pvm_bufinfo");
     info = pvm_upkint(&mydata, 1, 1);
     if (info < 0)
       pvm_perror("calling pvm_upkint");
     if (mydata != tid)
       printf("This should not happen!\n");
       printf("Length %d, Tag %d, Tid t%x\n", len, tag, tid);
  }
  pvm_exit();
  return 0;
}
 /* 자식처리부분 */
 info = pvm_initsend(PvmDataDefault);
 if (info < 0) {
    pvm_perror("calling pvm_initsend");
    pvm_exit();
```

```
        return -1;
    }
    info = pvm_pkint(&mytid, 1, 1);
    if (info < 0) {
          pvm_perror("calling pvm_pkint");
          pvm_exit(); return -1;
     }
    info = pvm_send(myparent, JOINTAG);
    if (info < 0) {
         pvm_perror("calling pvm_send");
         pvm_exit();
         return -1;
     }
    pvm_exit();
    return 0;
}
```

```
% fork-join
t10001c t40149 tc0037
Length 4, Tag 11, Tid t40149
Length 4, Tag 11, Tid tc0037
Length 4, Tag 11, Tid t10001c
% fork-join 4
t10001e t10001d t4014b tc0038
Length 4, Tag 11, Tid t4014b
Length 4, Tag 11, Tid tc0038
Length 4, Tag 11, Tid t10001d
Length 4, Tag 11, Tid t10001e
```

그림 3.1 fork-join 프로그람의 출구

그림 3.1은 fork-join프로그람의 실행결과를 보여주고있다. 통보가 접수되는 순서는 결정하지 못한다. 어미프로쎄스가 통보를 접수하는 기본고리는 먼저 들어온 통보를 먼저 처리하는 원칙이므로 출구하는 순서도 어미에게 자식처리의 통보가 도착하는 순서로 된다.

3.2 스칼라적계산

여기서는 단순한 포트란프로그람 PSDOT를 고찰한다. 이 프로그람은 스칼라적을 계산하는 프로그람이다. 이 프로그람은 배렬 X, Y의 스칼라적을 계산한다.

먼저 PSDOT는 PVMFMYTID()와 PVMFPARENT()를 호출한다.

PVMFPARENT()함수는 만일 과제가 다른 병렬가상기계과제로부터 생성된 것이 아니라면 PVMNOPARENT를 돌려준다.

만일 이 경우라면 PSDOT는 주프로그람으로 되며 PSDOT의 다른 자식프로그람들을 생성해야 한다. 다음에 PSDOT는 리용자로부터 프로쎄스의 개수와 계산해야 할 벡토르의 크기를 받는다.

생성된 매개 자식처리들은 배렬 X, Y의 n/nproc개의 원소들을 어미로부터 받는다. 우에서 n은 벡토르의 길이이며 nproc는 계산에 참가하는 프로쎄스들의 개수이다. 만일 nproc가 n을 균등하게 나누지 못한다면 나머지 원소들에 대한 스칼라적은 주프로그람이 계산한다.

부분프로그람 SGENMAT는 X, Y의 값들을 우연적으로 생성한다. 다음에 PSDOT는 n-1개의 자식프로쎄스들을 생성하고 새롭게 생성된 자식과제들에 X와 Y의 부분요소들을 보낸다.

통보에는 부분배렬들의 길이가 들어있다.

자식프로쎄스들을 생성하고 부분벡토르들을 보낸 다음에 어미프로쎄스는 X와 Y배렬에서 자기에게 해당한 부분을 계산한다. 어미프로쎄스는 그 다음 자식프로쎄스들로부터 부분적인 스칼라적계산결과들을 받는다.

어미프로쎄스는 모든 국부적인 스칼라적들을 접수하고 대역적인 스칼라적으로 그것들을 합친 다음에 전체적인 스칼라적을 계산한다.

만일 PSDOT프로그람이 자식처리라면 그는 먼저 어미로부터 배렬X, Y의 부분배렬들을 포함하고있는 통보를 받는다. 이 부분배렬들에 대한 스칼라적을 계산한 다음에 결과를 어미에게 돌려준다.

병렬가상기계 실례프로그람 PSDOT.F

```
* PSDOT 는 벡토르 X,  Y의 병렬스칼라적을 계산한다.*
* .. 외부부분루틴들 ..
EXTERNAL  PVMFMYTID,  PVMFPARENT,  PVMFSPAWN,  PVMFEXIT,
                PVMFINITSEND
EXTERNAL   PVMFPACK,   PVMFSEND,   PVMFRECV,   PVMFUNPACK,
                SGENMAT
*
* .. 외부기능들 ..
INTEGER ISAMAX
REAL SDOT
EXTERNAL ISAMAX, SDOT
*
* .. 고유함수들 ..
INTRINSIC MOD
*
* .. 파라메터들 ..
INTEGER MAXN
PARAMETER ( MAXN = 8000 )
INCLUDE  'fpvm.h'
*
* .. 벡토르 ..
INTEGER N, LN, MYTID, NPROCS, IBUF, IERR
INTEGER I, J, K
REAL LDOT, GDOT
```

```
*
* .. 배렬 ..
INTEGER TIDS(0:63)
REAL X(MAXN), Y(MAXN)
*
* PVM에 등록, 자기자신과 어미의 과제 ID번호를 얻는다.
*
CALL PVMFMYTID( MYTID )
CALL PVMFPARENT( TIDS(0) )
*
* 어미 프로쎄스
*
IF ( TIDS(0) .EQ. PVMNOPARENT ) THEN
*
* 시동정보를 얻는다.
*
WRITE(*, *) '얼마나 많은 프로쎄스가 참가하는가 (1-64)?'
READ(*, *) NPROCS
WRITE(*, 2000) MAXN
READ(*, *) N
TIDS(0) = MYTID
IF ( N .GT. MAXN ) THEN
WRITE(*, *)   'N은 지내 크다. 이 경우에 실행하기 위하여 파라메터
              MAXN을 증가시키시오.'

STOP
END IF
*
* LN은 국부적으로 처리해야 할 스칼라적요소들의 개수
*
```

```
J = N / NPROCS
LN = J + MOD(N, NPROCS)
I = LN + 1
*
* X 와 Y를 자유으로 생성한다.
*
CALL SGENMAT( N, 1, X, N, MYTID, NPROCS, MAXN, J )
CALL SGENMAT( N, 1, Y, N, I, N, LN, NPROCS )
*
* 모든 자식처리들에 대하여 반복한다.
*
DO 10 K = 1, NPROCS-1
*
* 프로쎄스들을 생성하고 오유검사
*
CALL PVMFSPAWN( 'psdot', 0, 'anywhere', 1,
                TIDS(K), IERR )
IF (IERR .NE. 1) THEN
  WRITE(*, *) 'ERROR, 프로쎄스를 생성할수 없습니다. #', K,
 CALL PVMFEXIT( IERR )
STOP
END IF
*
* 시동정보를 보낸다.
*
CALL PVMFINITSEND( PVMDEFAULT, IBUF )
CALL PVMFPACK( INTEGER4, J, 1, 1, IERR )
CALL PVMFPACK( REAL4, X(I), J, 1, IERR )
CALL PVMFPACK( REAL4, Y(I), J, 1, IERR )
CALL PVMFSEND( TIDS(K), 0, IERR )
```

```
I = I + J
10 CONTINUE
*
*스칼라적계산을 위해 어미처리가 처리해야 할 부분을 얻는다.
*
GDOT = SDOT( LN,  X,  1,  Y,  1 )
*
* 국부적인 스칼라적을 접수하고
* 그것을 대역적인 스칼라적에 더한다.
*
DO 20 K = 1, NPROCS-1
CALL PVMFRECV( -1, 1, IBUF )
CALL PVMFUNPACK( REAL4, LDOT, 1, 1, IERR )
GDOT = GDOT + LDOT
20 CONTINUE
*
* 결과출구
*
WRITE(*,  *) ' '
WRITE(*,  *)  '<x,  y> = ' ,  GDOT

LDOT = SDOT( N,   X,  1,  Y,  1 )
WRITE(*,  *)  '<x,  y> : sequential dot product. <x,  y>^ :'  //
$  'distributed dot product.'
WRITE(*,  *)  '| <x,  y> - <x,  y>^ | =' ,  ABS(GDOT - LDOT)
WRITE(*,  *)  '실행이 완성되였습니다.'
*
* 자식프로쎄스의 실행부분
*
ELSE
```

```
*
* 시동정보 접수
*
CALL PVMFRECV( TIDS(0),  0,  IBUF )
CALL PVMFUNPACK( INTEGER4, LN, 1, 1, IERR )
CALL PVMFUNPACK( REAL4, X, LN, 1, IERR )
CALL PVMFUNPACK( REAL4, Y, LN, 1, IERR )
*
* 국부적인 스칼라적을 계산하고 그것을 어미에게 보낸다.
*
LDOT = SDOT( LN, X, 1, Y, 1 )
CALL PVMFINITSEND( PVMDEFAULT, IBUF )
CALL PVMFPACK( REAL4, LDOT, 1, 1, IERR )
CALL PVMFSEND( TIDS(0), 1, IERR )
END IF
*
CALL PVMFEXIT( 0 )
*
1000 FORMAT(I10, ' 프로쎄스르를 성과적으로 생성하였습니다. #', I2,
', TID =', I10)
2000 FORMAT( 'Enter the length of vectors to multiply (1-' , I7,
'):' )
STOP
*
* PSDOT 프로그람의 끝
*
END
```

3.3 failure 프로그람

failure실례프로그람은 리용자가 과제들을 중지하는 방법, 과제들이 완료했는가 혹은 실패하였는가를 찾는 방법에 대하여 서술한다. 이 실례에서 우리는 이전의 실례에서와 마찬가지로 여러개의 과제들을 생성한다. 이 과제들중의 하나는 어미과제에 의하여 중지된다.

우리는 과제가 중지된것을 찾는데 흥미를 가지고있으므로 과제들을 생성한 pvm_notify()를 호출한다. pvm_notify()호출은 병렬가상기계에게 어떤 과제가 중지된 경우에 이 지령을 호출한 과제에 통보를 보낸다. 여기서 우리는 모든 자식과제들에 흥미를 가지고있다.

통지문은 또한 새로운 처리기가 가상기계에 추가되거나 가상기계에서 삭제된 경우 과제에게 통지하기 위하여 리용한다. 이 기능은 프로그람이 현재의 체계에 동적으로 적응하려고 할 때 효과적으로 리용할수 있다.

통지를 요구한 후에 어미과제는 자식과제들중 하나를 중지한다. 이때 pvm_kill()은 과제id 변수로 지적된 과제를 중지한다.

어미과제는 생성된 과제들중 한개의 과제를 중지한 후 pvm_recv(-1, TASKDIED)를 리용하여 과제가 중지하였다는 통보를 기다린다. 중지된 과제의 과제id는 통지문통보에서 유일한 옹근수로 기억된다. 프로세스는 중지된 과제의 과제 id를 꺼내서 출구한다.

실례프로그람 : failure.c

```
/* 병렬가상기계서고를 참조하기 위한 정의와 원형들 */
#include <stdio.h>
#include <pvm.h>
#define MAXNCHILD 20 /* 생성할 과제들의 최대개수 */
#define TASKDIED 11
main(int argc, char *argv[])
{
int ntask = 3;  /*생성할 과제의 개수, 기정으로는 3 */
int info;       /* 귀환코드 */
```

```
int mytid;     /* 과제의 과제 id */
int myparent;  /* 어미과제의 과제id */
int child[MAXNCHILD]; /* 자식과제들의 과제 id배렬 */
int i,  deadtid;
int tid;
char *argv[5];

mytid = pvm_mytid();  /* 과제의 과제 id를 얻는다. */
                    /* 오유검사 */
if (mytid < 0)  {
   pvm_perror(argv[0]); /* 오유출구 */
   return -1;  /* 프로그람완료 */
 }
myparent = pvm_parent();  /* 어미과제의 과제 id를 얻는다.
     PvmNoParent가 아닌 오유가 발생하면 오유처리 */
if ((myparent < 0) && (myparent != PvmNoParent)) {
     pvm_perror(argv[0]);
     pvm_exit();
     return -1;
 }
/* 만일 어미과제가 없다면 이 프로그람이 어미과제로 된다. */
if (myparent == PvmNoParent) {
/* 생성할 과제의 개수를 결정 */
if (argc == 2) ntask = atoi(argv[1]);
if ((ntask < 1) || (ntask > MAXNCHILD)) {
     pvm_exit();
     return 0;
 }
/* 자식과제들을 생성 */
info = pvm_spawn(argv[0], (char **)0,  PvmTaskDebug,
```

```
                (char *)0, ntask, child);
/* 과제가 성과적으로 생성되였는가를 검사 */
if (info != ntask) {
      pvm_exit();
      return -1;
}
/* tids 출구 */
for (i = 0; i < ntask; i++) printf("t%x\t", child[i]); putchar('\n');
/* 자식과제가 중지되였을 때 통지할것을 요구한다. */
info = pvm_notify(PvmTaskExit, TASKDIED, ntask, child);
if (info < 0) {
       pvm_perror("notify");
       pvm_exit();
       return -1;
 }
info = pvm_kill(child[ntask/2]);
if (info < 0)  {
       pvm_perror("kill");
       pvm_exit();
       return -1;
}
/* 통지를 기다린다. */
info = pvm_recv(-1, TASKDIED);
if (info < 0) {
      pvm_perror("recv");
      pvm_exit();
      return -1;
}
info = pvm_upkint(&deadtid, 1, 1);
if (info < 0)
```

```
  pvm_perror("calling pvm_upkint");
printf("Task t%x has exited.\n", deadtid);
printf("Task t%x is middle child.\n", child[ntask/2]);
pvm_exit();
return 0;
}
/* 자식과제 처리부분 */
sleep(63);
pvm_exit();
return 0;
}
```

3.4 행렬곱하기

다음 실례프로그람은 행렬곱하기알고리듬을 서술한다. 프로그람 mmult는 이 절의 마지막에서 찾을수 있다. mmult프로그람은 C = A*B를 계산한다. 여기서 C, A, B는 모두 바른행렬이다.

단순하게 하기 위해 풀이를 계산하는데 m*m개의 과제들이 참가한다고 가정한다. 매개 과제는 결과행렬 C의 부분블로크를 계산한다.

블로크의 크기와 m의 값은 지령행변수로 주어진다고 가정한다. 행렬 A와 B는 m^2개의 과제들에 분배된 블로크들로 기억된다. 프로그람에 대하여 구체적으로 보기전에 먼저 알고리듬을 서술한다.

우리는 m*m개의 과제그물을 가정한다.

매개 과제(tij : 여기서 0<i, j<m)는 초기에 블로크 Cij, Aij, Bij 를 포함하고있다. 알고리듬의 첫단계에서 대각선상에 있는 과제들(tij: 여기서 i = j)은 블로크 Aii 를 i째행의 다른 모든 과제들에게 보낸다. Aii의 전송후에 모든 과제들은 Aii*Bij을 계산하고 결과를 Cij에 더한다.

다음 단계에서 행렬 B의 렬블로크들이 회전된다. 즉 tij는 블로크 B에서 자기에게 해당한 부분을 t(i-1)j에 보낸다(과제 tij는 블로크 B에서 자기에게 해당한 부분을 t(m-1)j에 보낸다.).

이제 과제들은 첫 단계로 돌아간다. Ai(i+1)는 행 i에 있는 모든 과제들에 방송되며 알고리듬은 계속된다. M번 반복한 후에 C행렬은 A*B를 포함하고있으며 B행렬은 회전되였다.

병렬가상기계에서는 과제들사이에 호상 통신하는데는 그 어떤 제한이 없다. 그러나 이 프로그람에 대해서는 과제들이 2차원그물구조로 결합되여있는것처럼 생각한다.

과제들을 렬거하기 위하여 매 과제는 그룹 mmult에 결합된다. 그룹 id들은 과제들을 그물에 배치하기 위하여 리용된다.

그룹에 처음 결합하는 과제는 그룹번호 0을 가진다.

mmult프로그람에서 그룹번호가 0인 과제는 다른 과제들을 창조하며 행렬곱하기에 필요한 변수들을 과제들에 보낸다. 변수들은 m과 bklsize이다.

즉 블로크개수의 두제곱뿌리와 블로크의 크기이다. 모든 과제들이 창조되고 변수들이 전송된 다음에 pvm_barrier()가 호출되여 과제들 모두가 그룹에 결합하게 한다.

장벽동기화를 하지 않으면 과제가 그룹에 결합되지 않았기때문에 후에 pvm_gettid()를 호출하는 경우에 오유가 생긴다. 장벽동기화를 한 다음 같은 행에 있는 다른 과제들의 과제id 들을 배렬 myrow에 보관한다.

이것은 행에 있는 모든 과제들에 대하여 그룹id들을 계산하고 대응하는 그룹 id에 대한 과제id를 병렬가상기계에 문의하여 진행한다. 다음에 malloc()를 리용하여 행렬에 대한 블로크들을 할당한다. 실제적인 응용프로그람에서는 행렬들이 이미 할당되여있다고 가정한다.

다음에 프로그람은 그가 계산할 행렬 C의 행과 렬을 계산한다. 이것은 그룹 id의 값에 기초하고있다. 그룹의 범위는 0부터 m-1이다. 따라서 (mygid/m)의 옹근수부분은 과제의 행을 주며 (mygid mod m)은 렬을 준다. 류사한 배치수법을 리용하여 우리는 그물에서 우와 아래에 있는 과제들의 그룹 id를 직접 계산한다.

다음에 InitBlock()를 호출하여 블로크들을 초기화한다. 이 함수는 행렬 A를 란수값으로 B를 단위행렬로, C를 0으로 초기화한다. 이것은 A = C임을 검사함으로써 프로그람의 끝에서 계산을 검증한다.

마지막으로 기본순환에 들어가 행렬곱하기를 진행한다. 먼저 대각선에 있는 과제들은 행렬 A의 대응부분을 그 행에 있는 모든 과제들에 방송한다. 배렬 myrow는 실제로 방송하는 과제의 과제 id를 가지고있다. pvm_mcast()는 호출하는 과제를 제외하고 과제배렬에 있는 모든 과제들에 통보를 보낸다는것을 상기시킨다.

다중통신하는 과제와 블로크를 수신하는 과제는 대각선블로크와 과제에 있는 행렬B의 블로크에 대하여 A*B를 계산한다. 부분블로크들이 곱해지고 C블로크에 더해진 다음에 블로크 B를 수직으로 밀기한다.

특히 B블로크를 통보에 압축하고 그것을 우쪽과제에 보내며 과제id의 아래쪽으로부터 새로운 B블로크를 접수한다. 서로 다른 순환연산에 따라 A블로크들과 B블로크들을 보내는 통보표적들이 서로 다르다. 또한 pvm_recv()를 리용할 때 과제id들을 완전히 지적한다.

일단 계산이 완료되면 행렬곱하기가 C의 값을 정확히 계산했는가를 검증하여 A = C인가를 확인한다. 이 검사는 행렬곱하기함수안에서는 진행하지 않는다.

병렬가상기계는 과제가 중지되였음을 인식할 필요가 없으므로 pvm_lvgroup()를 호출할 필요가 없다. 그러나 pvm_exit()를 호출하기전에 그룹에서 탈퇴하는것이 좋다.

실례프로그람: mmult.c

```
/*
행렬곱하기
#include <stdio.h>
#include “pvm.h”

#define MAXNTIDS 100
#define MAXROW 10
#define ATAG 2
#define BTAG 3
#define DIMTAG 5
```

```
void InitBlock(float *a, float *b, float *c, int blk, int  row, int col)
{
int len, ind;
int i, j;
srand(pvm_mytid());
len = blk*blk;
for (ind = 0; ind < len; ind++) {
     a[ind] = (float)(rand()%1000)/100.0; c[ind] = 0.0;
 }
for (i = 0; i < blk; i++)
    for (j = 0; j < blk; j++) {
        if (row == col)
          b[j*blk+i] = (i==j)? 1.0 : 0.0;
        else
          b[j*blk+i] = 0.0;

    }
}

void BlockMult(float *c, float *a, float *b, int blk)
{
int i, j, k;
for (i = 0; i < blk; i++)
for (j = 0; j < blk; j ++)
for (k = 0; k < blk; k++)
c[i*blk+j] += (a[i*blk+k] * b[k*blk+j]);
}

main(int argc, char *argv[])
{
```

```
int ntask = 2;
int info;
int mytid, mygid;
int child[MAXNTIDS-1];
int i, m, blksize;
int myrow[MAXROW];
float *a, *b, *c, *atmp;
int row, col, up, down;

mytid = pvm_mytid(); /* 이 과제 id를 찾는다. */
pvm_setopt(PvmRoute, PvmRouteDirect);
if (mytid < 0) {      /* 오유검사 */
   pvm_perror(argv[0]);   /* 오유출구 */
   return -1;
 }
mygid=pvm_joingroup("mmult"); /* mmult그룹에 결합한다. */
if (mygid < 0) {
     pvm_perror(argv[0]); pvm_exit(); return -1;
}
/*그룹 id가 0이면 다른과제들을 창조한다. */
if (mygid == 0) {
/* 몇개의 과제들을 창조할것인가를 결정한다. */
if (argc == 3) {
      m = atoi(argv[1]);
      blksize = atoi(argv[2]);
 }
if (argc < 3) {
  fprintf(stderr, "usage: mmult m blk\n");
  pvm_lvgroup("mmult");
  pvm_exit();
```

```
      return -1;
     }
    ntask = m*m;
    if ((ntask < 1) || (ntask >= MAXNTIDS)) {
       fprintf(stderr, "ntask = %d not valid.\n", ntask);
       pvm_lvgroup("mmult");
       pvm_exit();
       return -1;
     }
    /* 과제가 하나밖에 없다면 과제를 창조할 필요가 없다. */
    if (ntask == 1) goto barrier;
    /* 아들과제 창조 */
    info = pvm_spawn("mmult", (char **)0, PvmTaskDefault,
                             (char *)0, ntask-1, child);
    if (info != ntask-1) {
       pvm_lvgroup("mmult");
       pvm_exit();
       return -1;
    }
    /* 행렬차수를 보낸다. */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&m, 1, 1);
    pvm_pkint(&blksize, 1, 1);
    pvm_mcast(child, ntask-1, DIMTAG);
}
    else {
    /* 행렬차수 접수 */
    pvm_recv(pvm_gettid("mmult", 0), DIMTAG);
    pvm_upkint(&m, 1, 1);
    pvm_upkint(&blksize, 1, 1);
```

```
ntask = m*m;
}
/* 과제들이 모두 그룹에 결합되게 한다.*/

barrier:
info = pvm_barrier("mmult", ntask);
if (info < 0)
   pvm_perror(argv[0]);
/* 같은 행에 있는 tid들을 찾는다. */
for (i = 0; i < m; i++)
myrow[i] = pvm_gettid("mmult", (mygid/m)*m + i);
/* 블로크를 위한 기억기 할당 */
a = (float *)malloc(sizeof(float)*blksize*blksize);
b = (float *)malloc(sizeof(float)*blksize*blksize);
c = (float *)malloc(sizeof(float)*blksize*blksize);
atmp = (float *)malloc(sizeof(float)*blksize*blksize);

if (!(a && b && c && atmp)) {
   fprintf(stderr, "%s: out of memory!\n", argv[0]);
   free(a);
   free(b);
   free(c);
   free(atmp);
   pvm_lvgroup("mmult");
   pvm_exit();
  return -1;
 }
/* 행과 렬을 계산한다. */
 row = mygid/m;
 col = mygid % m;
```

```
/* 이웃에 있는 우와 아래를 계산 */
up = pvm_gettid("mmult", ((row) ? (row-1):(m-1))*m+col);
down = pvm_gettid("mmult", ((row == (m-1)) ? col: (row+1)*m+col));
InitBlock(a, b, c, blksize, row, col); /* 블로크초기화 */
for (i = 0; i < m; i++)  {  /* 행렬곱하기 진행 */
/* 행렬 A의 블로크를 방송 */
 if (col == (row + i)%m) {
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(a, blksize*blksize, 1);
    pvm_mcast(myrow, m, (i+1)*ATAG);
    BlockMult(c, a, b, blksize);
 }
else {
     pvm_recv(pvm_gettid("mmult", row*m + (row +i)% m), (i+1)*ATAG);
     pvm_upkfloat(atmp, blksize*blksize, 1);
     BlockMult(c, atmp, b, blksize);
 }
/* B의 렬들을 회전 */
pvm_initsend(PvmDataDefault);
pvm_pkfloat(b, blksize*blksize, 1);
pvm_send(up, (i+1)*BTAG);
pvm_recv(down, (i+1)*BTAG);
pvm_upkfloat(b, blksize*blksize, 1);
}
/* 검사 */
for (i = 0 ; i < blksize*blksize; i++)
if (a[i] != c[i])
printf("Error a[%d] (%g) != c[%d] (%g) \n", i, a[i], i, c[i]);
printf("Done.\n");
free(a); free(b); free(c); free(atmp);
```

```
    pvm_lvgroup("mmult");
    pvm_exit();
    return 0;
}
```

3.5 1차원열전도방정식

여기서는 도선인 경우에 매질을 통하여 열확산이 진행되는 과정을 계산하는 병렬가상기계프로그람을 보여준다. 가는 도선상에서의 1차원열전도방정식을 고찰하자. 열전도방정식은 다음과 같이 표시된다.

$$\frac{\partial A}{\partial t} = \frac{\partial^2 A}{\partial x^2}$$

이 방정식의 불련속형식은 다음과 같다.

$$\frac{A_{i+1,j} - A_{i,j}}{\triangle t} = \frac{A_{i,j+1} - 2A_{i,j} + A_{i,j-1}}{\triangle x^2}$$

방정식의 풀이공식은

$$A_{i+1,j} = A_{i,j} + \frac{\triangle t}{\triangle x^2}(A_{i,j+1} - 2A_{i,j} + A_{i,j-1}).$$

초기조건과 경계조건은 아래와 같다:

모든 t에 대하여

$$A(t,0) = 0,\ A(t,1) = 0 \text{ for all } t$$
$$A(0,x) = \sin(\pi x) \text{ for } 0 \le x \le 1.$$

가상코드는 아래와 같다.

```
for i = 1: tsteps-1
    t = t+dt;
    a(i+1, 1)=0;
    a(i+1, n+2)=0;
    for j = 2: n+1;
        a(i+1, j)=a(i, j) + mu*(a(i, j+1)-2*a(i, j)+a(i, j-1));
    end;
    plot(a(i, j))
```

end

이 실례에서는 주-종속형식의 계산모형을 리용한다.

주프로그람인 heat.c는 5개의 종속프로그람인 heatslv프로그람을 생성한다. 종속프로그람들은 도선의 부분토막들에 대한 열확산과정을 병렬로 계산한다. 종속프로그람들은 매 단계에서 경계정보들을 교환한다. 이 경우에는 처리기들사이에 경계에서 도선의 온도값을 교환한다. 코드를 좀 더 구체적으로 보자.

heat.c에서 배렬 solution에는 매 단계에서의 열확산방정식에 대한 풀이를 가지고있다. 이 배렬은 프로그람의 끝에서 xgraph 형식으로 출구한다(xgraph는 자료를 찍는 프로그람이다.).

먼저 heatslv프로그람들을 생성한다. 다음에 초기자료모임들을 계산한다. 도선의 량끝에서는 초기온도값이 0으로 주어진다. 프로그람의 기본부분은 서로 다른 델타 t에 대하여 4번 실행한다.

계수기를 리용하여 매 단계에서 걸린 시간을 계산한다.

초기자료모임들은 heatslv과제들에 보낸다. 초기자료모임과 함께 왼쪽 및 오른쪽 과제들의 과제id를 보낸다. heatslv과제들은 이것을 리용하여 경계정보들을 교환한다. (선택적으로 병렬가상기계그룹호출을 리용하여 과제토막들을 도선에 배치할수 있다. 그룹호출을 리용하여 과제id들을 종속과제들에 보내는 문제를 피할수 있다.).

초기자료를 보낸 후에 주프로쎄스는 결과를 기다리기만 한다. 결과가 도착하면 그것을 풀이행렬에 통합하고 계산시간을 구하며 풀이를 xgraph에 출구한다.

일단 4단계에 따르는 자료를 계산하고 보관하면 주프로그람은 계산시간을 출구하고 종속프로쎄스들을 완료한다.

실례프로그람 : heat.c

/*

heat.c는 병렬가상기계를 리용하여 단순한 열확산방정식을 푼다. 한개의 주프로그람과 5개의 종속프로그람을 리용한다.

주프로그람은 자료들을 설정하고 그것들을 종속프로그람에 보내며 결과를 기다린다.

```
*/
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "pvm.h"

#define SLAVENAME "heatslv"
#define NPROC 5
#define TIMESTEP 100
#define PLOTINC 10
#define SIZE 1000
int num_data = SIZE/NPROC;

main()
{
  int mytid, task_ids[NPROC], i, j;
  int left, right, k, l;
  int step = TIMESTEP;
  int info;
  double init[SIZE], solution[TIMESTEP][SIZE];
  double result[TIMESTEP*SIZE/NPROC], deltax2;
  FILE *filenum;
  char *filename[4][7];
  double deltat[4];
  time_t t0;
  int etime[4];
  filename[0][0] = "graph1";
  filename[1][0] = "graph2";
```

```
  filename[2][0] = "graph3";
  filename[3][0] = "graph4";
  deltat[0] = 5.0e-1;
  deltat[1] = 5.0e-3;
  deltat[2] = 5.0e-6;
  deltat[3] = 5.0e-9;

mytid = pvm_mytid();   /* pvm에 등록 */
info  =  pvm_spawn(SLAVENAME,  (char  **)0,  PvmTaskDefault,  "",
                        NPROC, task_ids);
/* 초기자료모임 창조 */
for (i = 0; i < SIZE; i++)
init[i] = sin(M_PI * ( (double)i / (double)(SIZE-1) ));
init[0] = 0.0;
init[SIZE-1] = 0.0;
/* delta t의 서로 다른 값들에 대하여 프로그람을 4번 실행 */
for (l = 0; l < 4; l++) {
 deltax2 = (deltat[l]/pow(1.0/(double)SIZE, 2.0));
time(&t0);  /* 시동시간 계산 */
etime[l] = t0;
/* 초기자료들을 종속프로쎄스들에 보낸다. */
/* 경계자료들을 교환하는데 필요한 이웃정보들을 포함 */
for (i = 0; i < NPROC; i++) {
    pvm_initsend(PvmDataDefault);
    left = (i == 0) ? 0 : task_ids[i-1];
    pvm_pkint(&left,  1,  1);
    right = (i == (NPROC-1)) ? 0 : task_ids[i+1];
    pvm_pkint(&right,  1,  1);
    pvm_pkint(&step,  1,  1);
    pvm_pkdouble(&deltax2,  1,  1);
```

```
    pvm_pkint(&num_data, 1, 1);
    pvm_pkdouble(&init[num_data*i], num_data, 1);
    pvm_send(task_ids[i], 4);
 }
/* 결과 기다림 */
for (i = 0; i < NPROC; i++) {
  pvm_recv(task_ids[i], 7);
  pvm_upkdouble(&result[0], num_data*TIMESTEP, 1);
  for (j = 0; j < TIMESTEP; j++)  /* 풀이 갱신*/
   for (k = 0; k < num_data; k++)
     solution[j][num_data*i+k] = result[wh(j, k)];
}
/* 계수 중지*/
time(&t0);
etime[l] = t0 - etime[l];
/* 출구 */
filenum = fopen(filename[l][0], "w");
fprintf(filenum, "TitleText: Wire Heat over Delta Time: %e\n",
           deltat[l]);
fprintf(filenum, "XUnitText: Distance\nYUnitText:     Heat\n");
for (i = 0; i < TIMESTEP; i = i + PLOTINC) {
      fprintf(filenum, "\"Time index: %d\n", i);
     for (j = 0; j < SIZE; j++)
     fprintf(filenum, "%d %e\n", j, solution[i][j]);
     fprintf(filenum, "\n");
 }
  fclose (filenum);
}
/* 시간계수정보를 출구 */
printf("Problem size: %d\n", SIZE);
```

```
    for (i = 0; i < 4; i++)
    printf("Time for run %d: %d sec\n", i, etime[i]);
    /* 종속프로쎄스들을 중지*/
    for (i = 0; i < NPROC; i++)
     pvm_kill(task_ids[i]);
     pvm_exit();
}

    int wh(x,  y)
      int x,  y;
     {
       return(x*num_data+y);
     }
```

heatslv프로그람들은 도선을 통하여 진행되는 실제적인 열확산과정을 계산한다. 종속프로그람은 초기자료모임을 접수하고 이 자료모임에 기초하여 풀이를 반복계산하며(매 반복에서 경계정보들을 이웃들과 교환한다.) 결과들을 주프로그람에게 보낸다.

주과제들이 무한순환을 하지 않고 실행을 중지하라는 어떤 통보를 종속과제에 보낼수도 있다. 그러나 통보전송의 복잡성을 피하기 위하여 종속과제들에서는 무한순환을 리용하며 주프로그람으로부터 그 과제들을 중지하게 하였다.

매 단계에 대하여 또 매 계산단계에 들어가기전에 온도행렬의 경계값들이 교환된다. 먼저 왼쪽의 이웃과제들에 왼쪽경계요소들이 전송되며 오른쪽 이웃과제들로부터 접수된다. 마찬가지로 오른쪽경계요소들은 오른쪽의 이웃과제에게 보내지고 왼쪽이웃으로부터 접수된다.

실례 프로그람 : heatslv.c

```
/*
```

종속과제들은 주과제로부터 초기자료를 접수하고 이웃들과 경계자료들을 교환하며 도선에서의 열변화를 계산한다.

```
*/
#include <stdio.h>
#include "pvm.h"

int num_data;
main()
{
   int mytid,  left,  right,  i,  j;
   int timestep;
   double *init,  *A;
   double leftdata,  rightdata,  delta,  leftside,  rightside;

  mytid = pvm_mytid();
   /* 프로그람으로부터 자료접수 */
 while(1) {
   pvm_recv(right,  4);
   pvm_upkint(&left,  1,  1);
   pvm_upkint(&right,  1,  1);
   pvm_upkint(&timestep,  1,  1);
   pvm_upkdouble(&delta,  1,  1);
   pvm_upkint(&num_data,  1,  1);
   init = (double *)malloc(num_data*sizeof(double));
   pvm_upkdouble(init,  num_data,  1);
/* 초기자료를 작업배렬에 복사한다. */
   A = (double *)malloc(num_data * timestep * sizeof(double));
for (i = 0; i < num_data; i++)  A[i] = init[i];
/* 계산 */
for (i = 0; i < timestep-1; i++) {
```

```
    /* 이웃들과 경계정보를 교환 */
    /* 왼쪽으로 보내고 오른쪽에서 접수 */
     if (left != 0) {
           pvm_initsend(PvmDataDefault);
           pvm_pkdouble(&A[wh(i, 0)], 1, 1);
           pvm_send(left,  5);
     }
     if (right != 0) {
            pvm_recv(right,  5);
            pvm_upkdouble(&rightdata,  1,  1);
            pvm_initsend(PvmDataDefault);
            pvm_pkdouble(&A[wh(i, num_data-1)], 1, 1);
            pvm_send(right,  6);
     }
     if (left != 0) {
           pvm_recv(left,  6);
           pvm_upkdouble(&leftdata, 1, 1);
      }
      for (j = 0; j < num_data; j++) {
           leftside = (j == 0) ? leftdata : A[wh(i, j-1)];
           rightside = (j == (num_data-1)) ? rightdata :  A[wh(i, j+1)];
           if ((j==0)&&(left==0))
            A[wh(i+1, j)] = 0.0;
           else if ((j==(num_data-1))&&(right==0))
                   A[wh(i+1, j)] = 0.0;
           else
              A[wh(i+1, j)]=
            A[wh(i, j)]+delta*(rightside-2*A[wh(i, j)]+leftside);
     }
}
```

```
    /* 결과를 주프로그램에 보낸다. */
    pvm_initsend(PvmDataDefault);
    pvm_pkdouble(&A[0], num_data*timestep, 1);
    pvm_send(right,  7);
}
    pvm_exit();
}
    int wh(x,  y)
    int x,  y;
    {
      return(x*num_data+y);
    }
```